

高职高专计算机规划教材

Java 教程

郑阿奇 主 编

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本教程以 Java 最新的版本 Java SE Development Kit (JDK) 6 Update 10 为基础, 内容循序渐进、深入浅出, 精心设计每一个示例, 结构安排合理, 使读者准确把握 Java 的知识点。

本教程包括 Java 基础、习题、实验和习题答案四部分。本教程在讲解内容后紧跟实例, 每章的最后还配备了一个综合实例, 对已经学习的主要知识进行综合应用。实验部分通过实例引导读者进行学习, 并提出思考和练习。实例程序均通过上机调试。全书以开源软件 Eclipse 作为 Java 的集成开发环境, 使得编写、调试、运行 Java 程序变得更为简便。

本书专为高职高专设计, 可作为高职高专 Java 语言课程的教材, 也可作为 Java 自学者或者应用开发者的参考书。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目(CIP)数据

Java 教程 / 郑阿奇主编. —北京: 电子工业出版社, 2009.9

高职高专计算机规划教材

ISBN 978-7-121-09547-4

I. J... II. 郑... III. Java 语言—程序设计—高等学校: 技术学校—教材 IV. TP312

中国版本图书馆 CIP 数据核字 (2009) 第 168267 号

策划编辑: 赵云峰

责任编辑: 刘真平

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 21.75 字数: 556 千字

印 次: 2009 年 9 月第 1 次印刷

印 数: 4 000 册 定价: 32.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前 言

Java 是 Sun 公司开发的新一代编程语言，因其具有简单、面向对象、多线程、跨平台等特性而深受人们的欢迎。当前学习 Java 不仅是一种时尚，而且是一种潮流。

Java 教程以 Java 最新的版本 Java SE Development Kit (JDK) 6 Update 10 为基础，内容循序渐进、深入浅出，精心设计每一个示例，结构安排更为合理，使读者准确把握 Java 的知识点。实用教程一般在讲解一项内容后紧跟相关的实例演示。每章的最后还配备了一个综合实例，使学习者综合应用已经学过的主要知识。实验部分也是通过实例先引导读者进行学习，再提出思考练习。实例程序均通过上机调试，运行结果采用屏幕截图，避免代码与结果不一致的错误。全书以开源软件 Eclipse 作为 Java 的集成开发环境，这使得编写、调试、运行 Java 程序变得更为简便。

实际上，本教程不仅适合于教学，也非常适合于 Java 的各类培训和用 Eclipse 编程开发应用程序的用户学习和参考。只要阅读本书，结合上机操作进行练习，就能在较短的时间内基本掌握 Java 知识及其应用技术。

与本书配套的同步 PowerPoint 课件，可直接用于课堂教学。书中的源代码和 PowerPoint 课件，可从 <http://www.hxedu.com.cn> 或 <http://www.huaxin.edu.cn> 网站上免费下载。

本书专为高职高专设计，一些同志参加本书的基础工作，最后由南京师范大学郑阿奇统稿、定稿。

由于作者水平有限，不当之处在所难免，恳请读者批评指正。

意见建议邮箱：easybooks@163com。

编 者

2009 年 6 月

目 录

第 1 部分 Java 实用教程

第 1 章 Java 和 Eclipse 集成开发环境	1
1.1 Java 语言简介	1
1.2 第一个 Java 程序	2
1.3 Java 开发环境的搭建	3
1.4 Eclipse 集成开发环境	6
第 2 章 Java 语法基础	9
2.1 常量和变量	9
2.1.1 数据类型	9
2.1.2 标识符和关键字	9
2.1.3 常量	10
2.1.4 变量	12
2.1.5 类型转换	13
2.2 运算符和表达式	14
2.2.1 运算符	14
2.2.2 表达式	19
2.3 流程控制	19
2.3.1 分支语句	19
2.3.2 循环语句	21
2.3.3 流程跳转语句	25
2.4 方法与递归	26
2.4.1 方法	26
2.4.2 参数传递	26
2.4.3 递归	27
2.5 命名规范	28
2.6 注释语句	29
2.7 综合实例	29
第 3 章 Java 面向对象编程（上）	32
3.1 类的定义及成员变量初始化	32
3.1.1 类的定义	33
3.1.2 变量的初始化	33
3.2 创建对象	33
3.2.1 构造方法	33
3.2.2 默认构造方法	35
3.2.3 构造方法重载	36
3.2.4 普通方法重载	37
3.3 this 引用	38

3.4	静态成员	39
3.4.1	static 关键字	39
3.4.2	main()方法	41
3.4.3	类的初始化	42
3.5	package 与 import 语句	43
3.6	内部类	47
3.6.1	实例内部类	47
3.6.2	匿名类	49
第 4 章	Java 面向对象编程（下）	51
4.1	继承	51
4.1.1	继承的定义	51
4.1.2	初始化基类	52
4.1.3	方法的重写	53
4.1.4	super 关键字	56
4.2	对象的转型	56
4.3	多态	59
4.4	抽象类	61
4.5	接口	63
4.6	final 关键字	64
4.7	权限修饰符	65
4.7.1	类内部	66
4.7.2	同一个包的类	67
4.7.3	不同包的子类	68
4.7.4	通用性	69
4.8	综合实例：航班管理	69
第 5 章	常用类与异常处理	78
5.1	Object 类	78
5.1.1	equals()方法	78
5.1.2	hashCode()方法	80
5.1.3	toString()方法	80
5.2	字符串类	81
5.2.1	String 类	81
5.2.2	StringBuffer 类	83
5.3	包装类	85
5.4	Math 类	87
5.5	日期类	88
5.6	正则表达式	90
5.6.1	替换操作	95
5.6.2	Scanner 定界符	95

5.7	异常处理	96
5.7.1	异常的捕获与处理	97
5.7.2	声明抛出异常子句	99
5.7.3	抛出异常语句	100
5.7.4	自定义异常类	101
5.8	综合实例	103
第 6 章	数组与枚举	105
6.1	一维数组	105
6.2	多维数组	108
6.3	访问数组	111
6.4	数组实用类:Arrays	113
6.4.1	复制数组	113
6.4.2	数组排序	114
6.4.3	数组元素的查找	115
6.5	枚举	116
6.6	enum 的构造方法	117
6.7	综合实例	118
第 7 章	容器和泛型	121
7.1	Collection 与 Iterator	121
7.2	实用类 Collections	123
7.3	Set (集)	124
7.3.1	HashSet	124
7.3.2	TreeSet	126
7.4	List (列表)	129
7.4.1	ArrayList	129
7.4.2	LinkedList	130
7.5	Map (映射)	131
7.5.1	HashMap	132
7.5.2	TreeMap	133
7.6	泛型	135
7.7	通配符与受限通配符	137
7.8	综合实例	139
第 8 章	Java 输入/输出系统	142
8.1	字节流	142
8.1.1	文件输入流	143
8.1.2	文件输出流	144
8.2	过滤流	146
8.2.1	缓冲流类	146

8.2.2	数据流类	148
8.2.3	PrintStream 类	150
8.3	字符流	151
8.3.1	FileReader 和 FileWriter	152
8.3.2	BufferedReader 和 BufferedWriter	153
8.4	标准 I/O	154
8.5	File 类	156
8.6	综合实例	157
第 9 章	AWT 组件及应用	160
9.1	AWT 容器	160
9.1.1	Window 和 Frame	160
9.1.2	Panel	161
9.2	布局管理器	162
9.3	事件处理机制	164
9.3.1	AWT 事件与监听器	164
9.3.2	窗口事件	166
9.3.3	内部类实现监听接口	166
9.3.4	类自身实现监听接口	167
9.3.5	外部类实现监听接口	169
9.3.6	采用事件适配器	171
9.4	综合实例	172
第 10 章	Swing 组件及应用	177
10.1	窗口 JFrame	177
10.2	按钮	178
10.3	JTextField 与 JTextArea	180
10.4	JCheckBox 和 JRadioButton	183
10.5	菜单条 (JMenuBar)	186
10.6	弹出式菜单 (JpopupMenu)	189
10.7	综合实例	191
第 11 章	并发	195
11.1	线程的创建与启动	195
11.1.1	继承 java.lang.Thread 类	195
11.1.2	实现 Runnable 接口	197
11.2	线程的状态转换	198
11.3	线程调度	200
11.3.1	调整各个线程的优先级	200
11.3.2	线程让步	201
11.3.3	合并线程	202

11.4 后台线程 203

11.5 综合实例 204

第 12 章 综合实例 207

12.1 设计思路 207

12.2 汉诺塔上盘子模拟 207

12.3 汉诺塔上对象的定位及盘子的存放 208

12.4 创建汉诺塔及实现手动搬运盘子 209

12.5 自动搬运盘子 216

第 13 章 Java 网络编程 222

13.1 网络程序设计基础 222

13.1.1 TCP 和 UDP 222

13.1.2 端口和套接字 222

13.2 TCP 网络编程 223

13.2.1 InetAddress 类 225

13.2.2 TCP 通信程序 227

13.3 UDP 网络编程 230

13.3.1 UDP 通信程序 232

13.3.2 组播 234

13.4 URL 236

13.5 综合实例 238

第 14 章 JDBC 编程 243

14.1 SQL 语言 243

14.2 JDBC 245

14.3 MySQL 数据库 248

14.3.1 MySQL 服务器的安装 248

14.3.2 MySQL 服务器的配置 248

14.3.3 MySQL 的环境 250

14.4 访问数据库 251

14.4.1 加载并注册数据库驱动 252

14.4.2 建立到数据库的连接 253

14.4.3 访问数据库 254

14.5 JDBC 编程 256

14.6 批处理 261

14.7 事务处理 262

14.8 综合实例 264

第 2 部分 习题集

第 1 章 Java 和 Eclipse 集成开发环境 267

第 2 章 Java 语法基础 267

第 3 章 Java 面向对象编程（上） 267

第 4 章 Java 面向对象编程（下） 268

第 5 章 常用类与异常处理	268
第 6 章 数组与枚举	268
第 7 章 容器和泛型	268
第 8 章 Java 输入/输出系统	269
第 9 章 AWT 组件及应用	269
第 10 章 Swing 组件及应用	269
第 11 章 并发	270
第 13 章 Java 网络编程	270
第 14 章 JDBC 编程	270

第 3 部分 实 验

实验 1 Java 和 Eclipse 集成开发环境	271
实验目的	271
实验准备	271
实验内容	271
思考与练习题	273
实验 2 Java 语法基础	273
实验目的	273
实验准备	273
实验内容	273
思考与练习题	276
实验 3 Java 面向对象编程（上）	276
实验目的	276
实验准备	276
实验内容	277
思考与练习题	278
实验 4 Java 面向对象编程（下）	279
实验目的	279
实验准备	279
实验内容	279
思考与练习题	281
实验 5 常用类与异常处理	281
实验目的	281
实验准备	281
实验内容	282
思考与练习题	284
实验 6 数组与枚举	284
实验目的	284
实验准备	284
实验内容	284
思考与练习题	287

实验 7 容器和泛型..... 287

 实验目的..... 287

 实验准备..... 287

 实验内容..... 287

 思考与练习题..... 290

实验 8 Java 输入/输出系统..... 290

 实验目的..... 290

 实验准备..... 290

 实验内容..... 290

 思考与练习题..... 293

实验 9 AWT 组件及应用..... 293

 实验目的..... 293

 实验准备..... 293

 实验内容..... 294

 思考与练习题..... 298

实验 10 Swing 组件及应用..... 299

 实验目的..... 299

 实验准备..... 299

 实验内容..... 299

 思考与练习题..... 303

实验 11 并发..... 303

 实验目的..... 303

 实验准备..... 303

 实验内容..... 303

 思考与练习题..... 306

实验 12 综合实例..... 306

 实验目的..... 306

 实验准备..... 306

 实验内容..... 306

 思考与练习题..... 306

实验 13 Java 网络编程..... 306

 实验目的..... 306

 实验准备..... 307

 实验内容..... 307

 思考与练习题..... 310

实验 14 JDBC 编程..... 310

 实验目的..... 310

 实验准备..... 310

 实验内容..... 310

 思考与练习题..... 314

第 4 部分 习题答案

第 1 章	习题答案	315
第 2 章	习题答案	315
第 3 章	习题答案	317
第 4 章	习题答案	318
第 5 章	习题答案	320
第 6 章	习题答案	320
第 7 章	习题答案	323
第 8 章	习题答案	324
第 9 章	习题答案	325
第 10 章	习题答案	328
第 11 章	习题答案	331
第 13 章	习题答案	331
第 14 章	习题答案	333

第 1 部分 Java 实用教程

第 1 章 Java和Eclipse集成开发环境

1.1 Java语言简介

1991 年, Sun 公司开发了新一代编程语言 Java, 初衷是为家用消费类电子产品开发一个分布式代码系统。为了使整个系统与平台无关, 采用了虚拟机器码 (Virtual Machine Code) 方式, 虚拟机运行在一个解释器上, 每一个操作系统均有一个解释器。1995 年 3 月发布了 Java 的 Alpha1.0a2 版本, 1996 年 1 月发布了 Java 的第一个开发包 JDK v1.0, 1997 年 2 月发布了 Java 语言的开发包 JDK v1.1, 从此奠定了 Java 在计算机语言中的地位。1998 年 12 月, Sun 公司发布 Java 2 平台 JDK v1.2, 这是 Java 发展史上的里程碑。1999 年 6 月, Sun 公司重新组织 Java 平台的集成方法, 并将企业级应用平台作为 Java 发展方向, 主要有 3 个成员:

- J2ME——Java 2 Micro Edition, 用于嵌入式应用的 Java 2 平台。
- J2SE——Java 2 Standard Edition, 用于工作站、PC 的 Java 2 标准平台。
- J2EE——Java 2 Enterprise Edition, 可扩展的企业级应用的 Java 2 平台。

2004 年, J2SE1.5 发布, 为了表示这个版本的重要性, J2SE1.5 更名为 J2SE5.0。2005 年, JavaOne 大会召开, Sun 公司公开 JavaSE6, 此时, Java 的各种版本被更名, 取消其中的数字“2”, J2SE 更名为 Java SE, J2EE 更名为 Java EE, J2ME 更名为 Java ME。

Java 是一个广泛使用的网络编程语言, 它简单, 面向对象, 不依赖于机器的结构, 不受 CPU 和环境的限制, 具有可移植性、安全性, 并且提供了并发的机制, 具有很高的性能。此外, Java 还提供了丰富的类库, 使程序设计人员可以方便地建立自己的系统。

Java 有两种核心机制: 一种是 Java 虚拟机 (Java Virtual Machine), 另一种是垃圾收集机制 (Garbage Collection)。

1. Java虚拟机

Java 程序是如何做到“一次编译, 到处运行”的呢? 这正是通过 Java 虚拟机来实现的。JVM 可以理解成一个以字节码为机器指令的 CPU。首先, Java 编译程序将后缀名为.java 的 Java 源程序编译为 JVM 可执行的代码, 即后缀名为.class 的 Java 字节码文件, 如图 1.1 所示。运行 JVM 字节码的工作是由解释器来完成的。解释执行过程分代码的装入、代码的校验和代码的执行 3 步进行。装入代码的工作由“类装载器”完成, 类装载器负责装入一个程序运行需要的所有代码。字节码校验器负责代码的校验。每种类型的操作系统都有一种对应的 Java 虚拟机, Java 虚拟机屏蔽了底层操作系统的差异。所以 Java 程序能够做到“一次编译, 到处运行”。

2. 垃圾收集机制

垃圾回收器能够自动回收垃圾，即无用的对象所占据的内存空间被回收。在 C/C++语言中，这些工作由程序员负责，无疑增加了程序员的负担。而 Java 语言消除了程序员回收垃圾的责任：它提供一种系统级线程来跟踪存储空间的分配情况，并在 JVM 的空闲时，检查并释放那些可被释放的存储空间。在 Java 中，对象被创建后，就会在堆区中分配一块内存。当对象不再被程序引用时，它就变成一个垃圾，所占用的堆空间可以被回收，以便空间被后续的新对象所使用。Java 的垃圾回收器能断定哪些对象不再被引用，并且能够把它们所占据的堆空间释放出来。

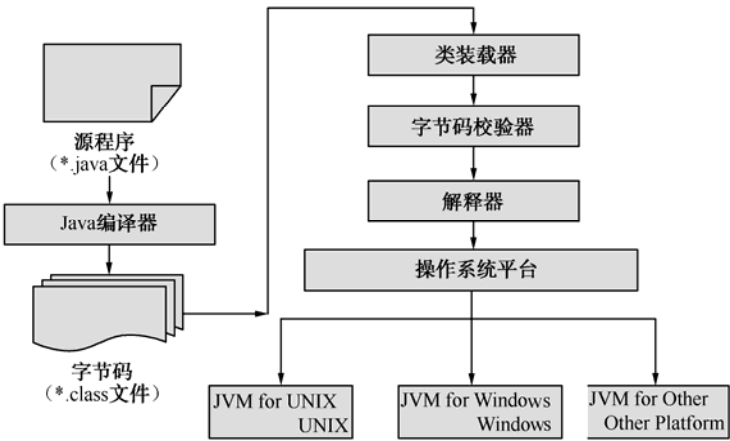


图 1.1 Java 程序执行流程

1.2 第一个Java程序

下面先看一个 Java 小程序，对 Java 编程有一个初步的认识。

【例 1.1】 求圆面积。

用文本编辑器（如 Windows 的记事本）编辑下列程序，文件名为 Area.java。

Area.java

```
/** 求圆的面积 */
public class Area {
    double pi = 3.1415;                // 定义变量 pi
    public static void main(String[] args) {
        double r,area;
        r = 3;
        area = pi * r * r;              // 求圆的面积
        System.out.println("圆的面积="+area); // 在屏幕上显示结果
    }
}
```

下面对程序进行简要说明。

(1) /**到*/之间的内容为注释。

(2) 保留字 `class` 声明一个类，其类名为 `Area`，保留字 `public` 表示它是一个公共类。类定义由花括号 `{}` 括起来。当编写一个 Java 源代码文件时，此文件通常被称为编译单元。每个编译单元都必须有一个后缀名 `.java`，而在编译单元内则可以有一个 `public` 类，该类的名称必须与文件的名称相同（包括大小写）。每个编译单元最多只能有一个 `public` 类，也可以没有，否则编译器就不会接受。

(3) 语句 `double pi = 3.1415;` 表示定义实型变量 `pi` 的值为 3.1415。

(4) 在该类中定义了一个 `main()` 方法，其中 `public` 表示访问权限，指明所有的类都可以使用这一方法；`static` 指明该方法是一个类方法，它可以通过类名直接调用；`void` 则指明 `main()` 方法不返回任何值。对于一个应用程序来说，`main()` 方法是必需的，而且必须按照如上的格式来定义。Java 解释器在没有生成任何实例的情况下，以 `main()` 方法作为入口来执行程序。Java 程序中可以定义多个类，每个类中可以定义多个方法，但是最多只能有一个公共类，`main()` 方法也只能有一个。

1.3 Java开发环境的搭建

要想编译和运行上面的程序，离不开Java的编译和运行环境。Sun公司提供了自己的一套Java开发环境，通常称为JDK（Java Development Kit），又称为J2SDK。目前最新的版本是Java SE Development Kit（JDK）6 Update 10，可以到Sun公司的网站下载。输入网址 `http://java.sun.com/javase/downloads/index.jsp`，进入第二栏，单击“Download”按钮，如图 1.2 所示。



图 1.2 选择操作系统

在“Platform”栏中选择 Windows，如果自己的平台是 Linux 操作系统，请选择 Linux，单击“Continue”按钮。之后进入下一屏，选中“Windows offline installation”单选按钮，单击“jdk-6u10-windows-i586-p.exe”项，JDK 就可以开始下载了。

下载完成后，双击可执行文件 `jdk-6u10-windows-i586-p.exe`，按照提示完成安装。这里 JDK 的安装路径改为“`C:\Java\jdk1.6.0_10\`”，如图 1.3 所示。

在 JDK 的安装包里带有 JRE 安装包，JRE 是 Java 运行时的环境。JRE 的安装路径改为“`C:\Java\jre6\`”，如图 1.4 所示。

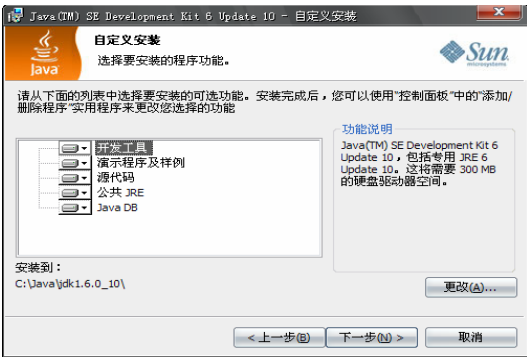


图 1.3 JDK 的安装

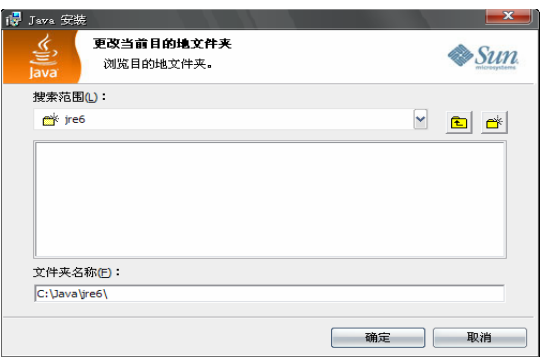


图 1.4 JRE 的安装

安装完成之后，还需要配置环境变量。在 Windows 系统中，在桌面上右击“我的电脑”，从打开的快捷菜单中选择“属性”命令，在打开的“系统属性”对话框中单击“高级”选项卡，如图 1.5 所示，单击“环境变量”按钮，打开“环境变量”对话框，如图 1.6 所示。

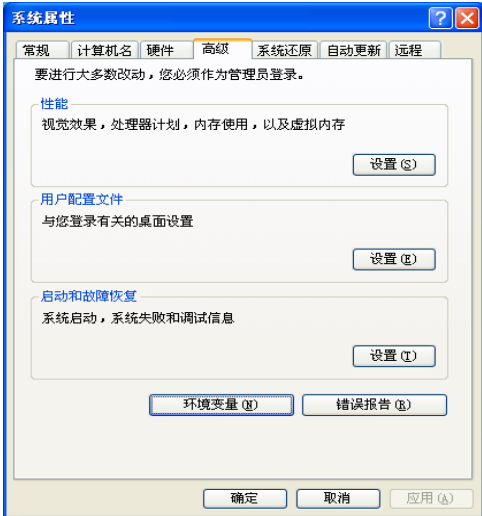


图 1.5 “系统属性”对话框



图 1.6 “环境变量”对话框

新建环境变量：在系统变量中双击“path”变量，在变量值栏内填入“C:\Java\jdk1.6.0_10\bin;”，在系统变量中再双击“CLASSPATH”变量，输入值为“.”，表示当前路径。如果变量值栏内还没有“CLASSPATH”变量，请新建“CLASSPATH”变量。最后，单击“确定”按钮，JDK 就配置完成并可以使用了。

下载的 JDK 开发工具包并没有包含 JDK API 文档，API 文档中提供了 JDK 中的类的完整使用说明，应下载下来以备随时查阅。API 文档的首页如图 1.7 所示。

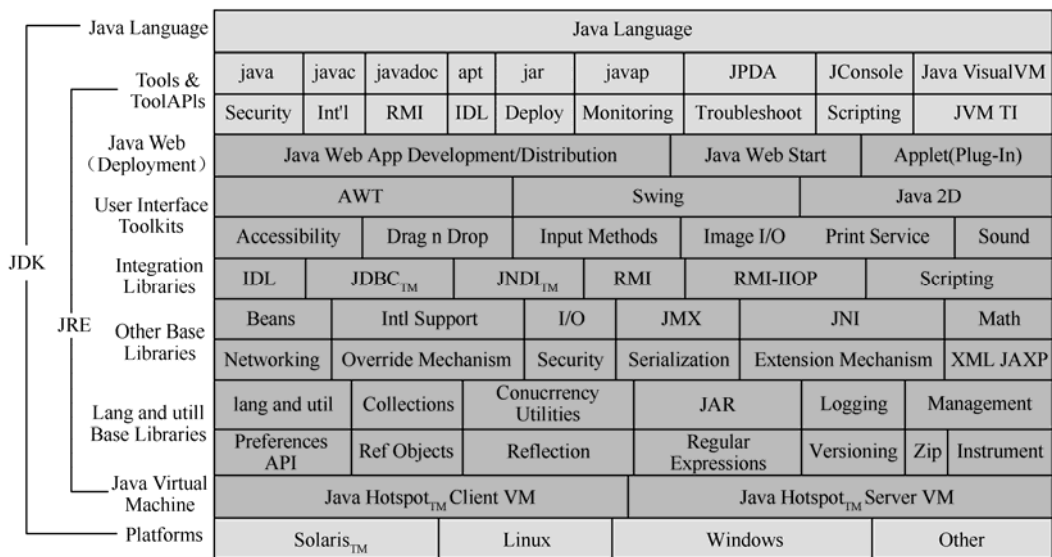


图 1.7 API 文档的首页

进入 JDK 的安装路径，将看到如表 1.1 所示的目录。

表 1.1 JDK 的常用目录结构

目 录	描 述
src 子目录	存放 Java 源文件
bin 子目录	存放 JDK 的工具程序
lib 子目录	存放 JAR 文件
demo 子目录	存放 Java 示范程序

JDK 主要包括以下内容。

- (1) Java 虚拟机: 负责解释和执行 Java 程序, Java 虚拟机可以运行在各种操作系统平台上。
- (2)JDK 类库: 提供了最基础的 Java 类库, 即各种实用类, 包括最常用的类库, 如 java.lang、java.io、java.util、javax.swing 和 java.sql 等。
- (3) 开发工具: 这些开发工具都是可执行程序, 主要包括 javac.exe (编译工具)、java.exe (运行工具)、javadoc.exe (生成 Javadoc 文档的工具) 和 jar.exe (打包工具) 等。

Java 的环境变量已配置好, 现在就可以编写 Java 程序了。用记事本就可以编写 Java 程序。首先建立一个名为“Area.java”的源文件, 存放在 d 盘根目录中。

打开命令提示符, 进入 d 盘根目录, 输入“javac Area.java”命令, 这时在 d 盘根目录下出现一个“Area.class”文件, 输入“java Area”命令。在控制台上显示出“圆的面积=28.2735”, 如图 1.8 所示。javac 命令把 Java 源文件编译成“.class”二进制文件, java 命令解释执行“.class”文件。



图 1.8 执行第一个 Java 程序

1.4 Eclipse集成开发环境

目前，编写 Java 程序普遍采用功能强大且免费的开发工具 Eclipse。可从 Eclipse.org 网站 <http://www.eclipse.org/downloads/> 下载最新的 Eclipse 发布版本，目前最新的稳定版本是 Eclipse 3.4。下载后，直接解压即可使用。解压后，在磁盘上生成一个 eclipse 文件夹，进入 eclipse 文件夹，双击 eclipse.exe 可执行文件，出现如图 1.9 所示的界面。

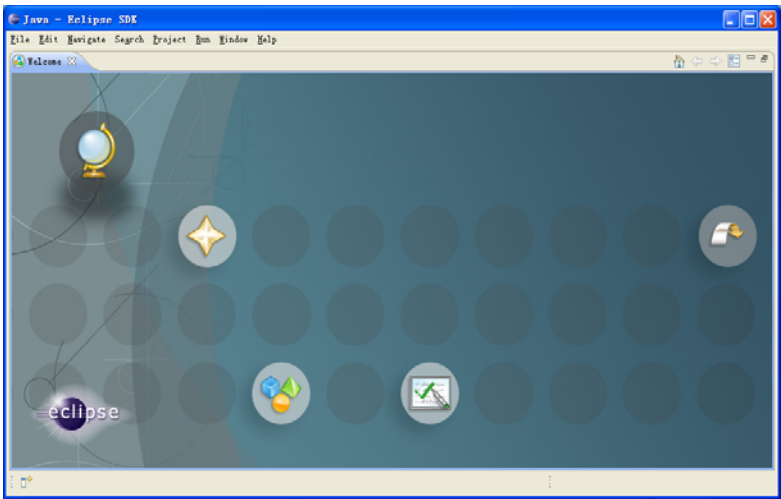


图 1.9 Eclipse 欢迎界面

为了使用方便，右击 eclipse.exe 文件，选择“发送到”→“桌面快捷方式”命令，之后在桌面上将出现 eclipse 的快捷方式。以后直接双击 eclipse 的快捷方式，即可启动 eclipse。现在就用 Eclipse 开发一个 Java 工程。开发一个基本的 Java 工程包括以下 3 个步骤。

1. 创建Java项目

进入图 1.10 所示界面，在工作台主窗口中，依次选择“File”→“New”→“Java Project”命令，打开新建项目向导，在“Project name”栏中输入项目名“MyProject_01”，其他选项默认。单击“Finish”按钮，项目创建成功，项目“MyProject_01”将出现在左边的 Navigator（导航器）中。

2. 创建Java包

在 Navigator 中右击项目 “MyProject_01”，选择 “New” → “Package”，如图 1.11 所示，在 “Name” 栏中输入包名 “org.circle”，单击 “Finish” 按钮完成包的创建。

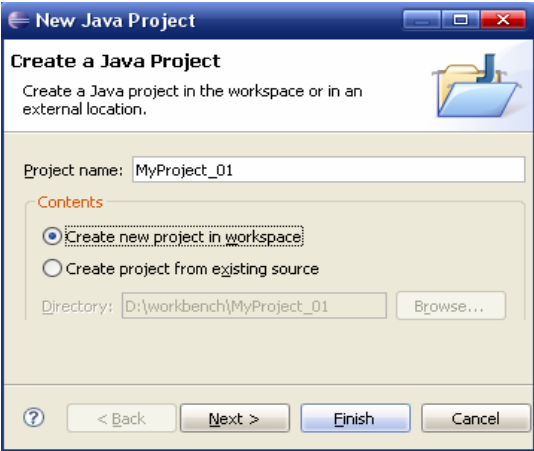


图 1.10 创建 Java 项目

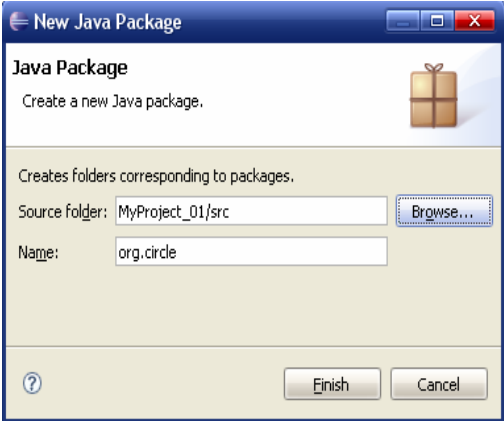


图 1.11 创建 Java 包

3. 创建Java类

右击项目 “MyProject_01” 的包 “org.circle”，选择 “New” → “Class”，如图 1.12 所示，在 “Name” 栏中输入类名 “Area”，单击 “Finish” 按钮完成类的创建。这时可以编写 Java 程序了。双击 “Area.java”，输入 “Area.java” 源程序，如图 1.13 所示，单击 “保存” 按钮。

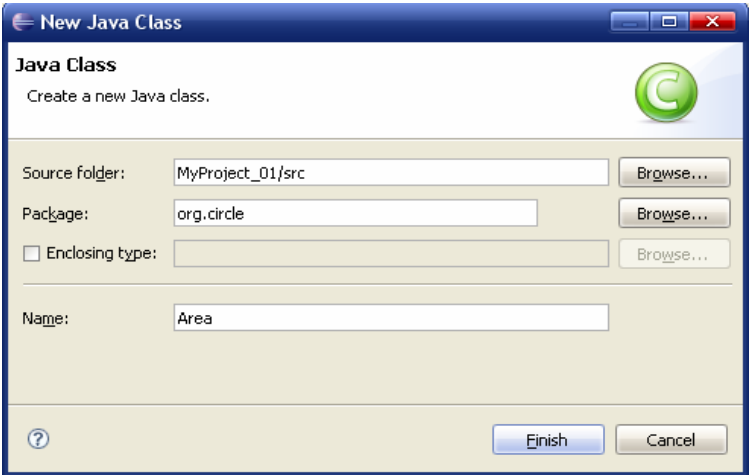


图 1.12 创建 Java 类

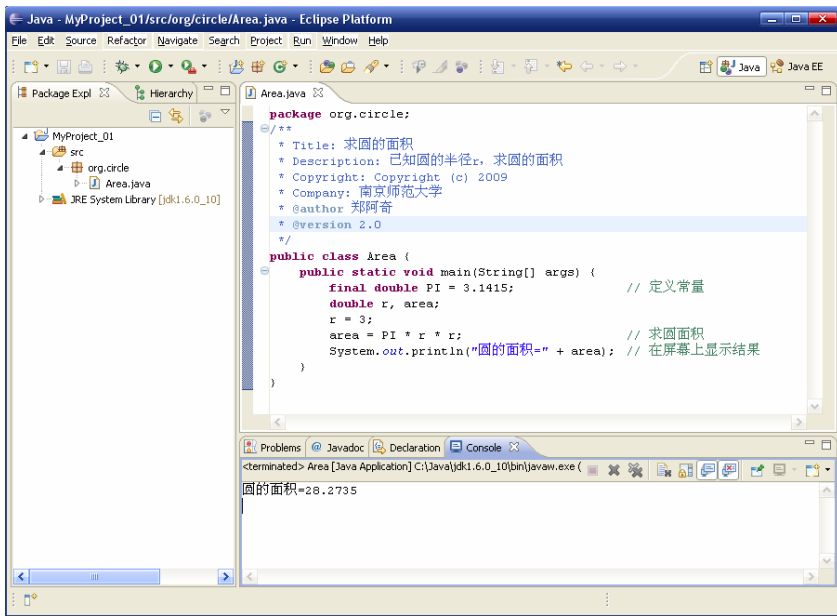


图 1.13 Eclipse 工作台

右击“Area.java”，选择“Run as”→“Java Application”命令，运行 Java 程序，将在控制台显示下面的结果：

圆的面积=28.2735

第 2 章 Java语法基础

程序由语句组成，语句经常使用数据类型、运算符、表达式等。Java 语言的数据类型、运算符与表达式等是从 C++语言简化而来的，简洁、高效。

2.1 常量和变量

Java 程序运行时值不可修改的数据称为常量，常量又分为字面常量（常数）与标识符常量两种。变量是程序运行时值发生改变的量。

2.1.1 数据类型

Java 是一种强类型的语言，这意味着所有变量都必须先明确定义其数据类型，然后才能使用。Java 语言的数据类型只有两类：基本数据类型与引用类型。基本数据类型的属性如表 2.1 所示。

表 2.1 基本数据类型的属性

数据类型	关键字	在内存占用的字节数	取值范围	默认值
布尔型	boolean	无明确规定	true、false	false
字节型	byte	1 个字节（8 位）	-128~127	(byte) 0
短整型	short	2 个字节（16 位）	-2 ¹⁵ ~2 ¹⁵ -1	(short) 0
整型	int	4 个字节（32 位）	-2 ³¹ ~2 ³¹ -1	0
长整型	long	8 个字节（64 位）	-2 ⁶³ ~2 ⁶³ -1	0L
字符型	char	2 个字节（16 位）	0~2 ¹⁶ -1	'\u0000'
单精度浮点型	float	4 个字节（32 位）	1.4013E-45~3.4028 E-45	0.0F
双精度浮点型	double	8 个字节（64 位）	4.9E-324~1.7977E+308	0.0D

基本数据类型共 8 种，其他为引用类型，例如，数组、字符串、类、接口、用户自定义类型等。所有基本数据类型的大小（所占用的字节数）都已明确规定好，在各种不同的平台上都保持一致，这一特性有助于提高 Java 程序的可移植性。

2.1.2 标识符和关键字

标识符是指程序中包、类、接口、变量或方法的名字的字符序列。Java 语言要求标识符必须符合以下命名规则。

- 标识符的首字符必须是字母、下画线“_”、美元符号“\$”。
- 标识符由数字（0~9）、大写字母（A~Z）、小写字母（a~z）、下画线“_”以及美元符号“\$”等组成，并且标识符的长度不受限制。
- 不能把关键字和保留字作为标识符。
- 标识符是大小写敏感的，例如，hello 与 Hello 是两个不同的标识符。

表 2.2 所示是一个标识符正误对照表，列举了一些合法的标识符和非法标识符。

表 2.2 标识符正误对照表

合法的标识符	不合法的标识符	说 明
HelloWorld	class	不能用关键字做标识符
_123	12.3b	标识符不能以数字开头
\$a123	Hello World	标识符中不能含有空格
Subject	Subject#	标识符中不能含有“#”

在命名 Java 标识符时，应注意“见名知意”。Java 中一些被赋予特定含义，具有专门用途的字符序列称为关键字，包括：

- 数据类型：boolean、byte、short、int、long、double、char、float、double。
- 包引入和包声明：import、package。
- 用于类和接口的声明：class、extends、implements、interface。
- 流程控制：if、else、switch、do、while、case、break、continue、return、default、while、for。
- 异常处理：try、catch、finally、throw、throws。
- 修饰符：abstract、final、native、private、protected、public、static、synchronized、transient、volatile。
- 其他：new、instanceof、this、super、void、assert、const*、enum、goto*、strictfp。

注意：打上“*”的关键字，Java 没有使用。

2.1.3 常量

Java 程序运行时值不可修改的量称为常量，它又分为字面常量（常数）与标识符常量两种。标识符常量实际上是一个变量，但它的值一旦初始化以后，就不允许再发生改变，因此标识符常量要先定义后使用，一般用于给一个常数取一个有意义的名字。字面常量即是 Java 源程序中表示的常数值，如 12.6、123、0x12、false、“hello”等，表示一个个具体的值。常量是有类型的，有如下 5 种。

1. 布尔型常量 (boolean)

布尔型常量值只有 true 或 false。 true 和 false 可以看成 Java 语言的关键字，不能挪作他用且必须要小写。true 表示“逻辑真”，false 表示“逻辑假”。

注意：不能认为“非零值或 1”是 true，“零值”是 false。

2. 整型常量 (int或long)

整型常量有十进制、八进制、十六进制 3 种表示法。

- 十进制：十进制整数，如 123、-48 等。
- 八进制：以数字 0 开头的八进制整数，如 017、-021 等。
- 十六进制：以 0x 或 0X 开头的十六进制整数，如 0x12a6、0XAB12、-0x1a0 等。

整型常量 (int) 在机器中占 32 位，即 4 个字节，故最大的整型常量是 2 147 483 647，该值由 Integer.MAX_VALUE 表示；最小的整型常量是-2 147 483 648，该值由 Integer.MIN_VALUE 表示。若程序中出现一个整型常量，其值超出上述范围，则会产生一个编译错误。为避免该错误，

标值的起点从 0 开始)。

注意：null 可以简单看成一个引用类型的常量值，表示不引用任何对象。在 Java 规范中，规定 null 是所谓 null 类型的唯一值，且规定 null 可转换到任何引用类型。

除了上边介绍的字面常量外，Java 中主要的是标识符常量。一个标识符常量是一个变量，一旦它的值初始化后，就再不能改变。它通常用于给一个常数取一个有意义的名字。标识符常量可以是任何类型，其定义格式是在变量定义的前面加上 final 保留字，如定义一个 double 型的常量 PI：final double PI = 3.14;。

按 Java 编码规范，常量名字通常用大写表示，若常量名由两个或两个以上单词组成，则单词间用下划线连接，例如，final int MAX_VALUE = 100;。

2.1.4 变量

与常量不同，变量是程序运行时值发生改变的量。变量对应着内存空间中的一个或几个单元，变量的值就存放在所对应的内存单元中。变量名就是给对应的内存单元取一个有意义的名称，这样在程序中，可以按变量名称来区分和使用这些内存单元。

1. boolean类型

boolean 类型的变量的取值只能是 true 或 false。在 Java 虚拟机内部，用 int 或 byte 类型来表示 boolean，用整数零表示 false，用任意一个非零整数来表示 true。只不过这些对程序员是透明的。

2. byte、short、int和long类型

byte、short、int 和 long 都是整数类型，并且都是有符号整数。在定义这 4 种类型的整型变量时，Java 编译器给它们分配的内存空间大小是不相同的。byte 类型占用的内存空间最小，为 1 个字节大小；long 类型占用的内存空间最大，占 8 个字节空间。如果一个整数值在某种整数类型的取值范围内，就可以把它直接赋给这种类型的变量，否则必须进行强制类型转换。例如，整数 13 在 byte 类型的取值范围（-128~127）内，因此可以把它直接赋给 byte 类型变量。例如，129 不在 byte 类型的取值范围内，若想赋给 byte 类型变量，则必须进行强制类型转换。

```
byte b = (byte)129 //变量 b 的取值为-127
```

以上代码中的“(byte)”表示把 129 强制转换为 byte 类型。129 的二进制数据形式为：

0000 0000 0000 0000 0000 0000 1000 0001。“(byte)”运算符截取后 8 位，为 1000 0001，其值为-127。

如果在整数后面加上后缀：大写“L”或小写“l”，就表示它是一个 long 类型的整数。

3. char类型与字符编码

char 是字符类型，Java 语言对字符采用 Unicode 字符编码，这是一种双字节编码，表示字符范围‘\u0000’～‘\uffff’。由于计算机只能存储二进制数据，因此必须为各个字符进行编码。所谓字符编码，是指用一串二进制数据来表示特定的字符。Unicode 字符编码由国际 Unicode 协会编制，收录了全世界所有语言文字中的字符，是一种跨平台的字符编码。

4. float和double类型

float 和 double 类型都遵循 IEEE 754 标准，该标准分别为 32 位和 64 位浮点数规定了二

进制数据表示形式。IEEE 754 采用二进制数据的科学记数法来表示浮点数。对于 float 浮点数，用 1 位表示数字的符号，用 8 位来表示指数（底数为 2），用 23 位来表示尾数。对于 double 浮点数，用 1 位表示数字的符号，用 11 位来表示指数（底数为 2），用 52 位来表示尾数。

2.1.5 类型转换

每一个表达式都有一个类型。当使用表达式时，表达式所处的上下文要求的类型与表达式类型不一致时，就会发生类型转换。常见的发生类型转换的上下文有：赋值时类型转换；方法调用时类型转换；强制类型转换；字符串类型转换；数值类型提升。

当上下文中要求类型转换时，转换应遵循什么规则呢？对于 Java 基本数据类型，类型转换有以下几种。

1. 宽转换

所谓的宽转换又称为自动类型转换或隐式转换。转换规则如下。

- (1) byte 可直接转换到 short、int、long、float、double。
- (2) short 可直接转换到 int、long、float、double。
- (3) char 可直接转换到 int、long、float、double。
- (4) int 可直接转换到 long、float、double。
- (5) long 可直接转换到 float、double。
- (6) float 可直接转换到 double。

注意：① byte、short、int 都是有符号的数，因而宽转换时（如转换到 long）进行符号位扩展。char 实际上是 0~65535 之间的无符号的数，其符号位可认为是 0，因而转换到 int 或 long 时，永远是用 0（二进制位零）进行扩展。例如，int i = (char) (byte)(-1)；i 的值为 65 535。

② int 转换到 float 或 long 转换到 double，很可能会造成精度丢失。例如，int big = 1234567891；float f = big；此时 f 中小数点之后已不能精确表示 891 了，即精度丢失了。

2. 窄转换

窄转换的转换规则如下。

- (1) short 可直接转换到 byte、char。
- (2) char 可直接转换到 byte、short。
- (3) int 可直接转换到 byte、short、char。
- (4) long 可直接转换到 byte、short、char、int。
- (5) float 可直接转换到 byte、short、char、int、long。
- (6) double 可直接转换到 byte、short、char、int、long、float。

注意：① 窄转换在大多数情况下会丢失信息。当 int 窄转换到 byte 时，是丢弃掉 int 的高 3 个字节（24 位），将最低的 1 个字节（8 位）放入 byte 中。因此，大多数情况下窄转换要求程序员自己明确地指明。

② char 窄转换到 short，将 char 的两个字节（16 位）直接放入到 short 中。虽然 char 与 short 都是 16 位，但窄转换到 short 时，其结果可能会由正数变成一个负数。

3. 宽窄同时转换

宽窄同时转换发生在 byte 转换到 char 期间。其转换过程为：先将 byte 宽转换到 int，再将 int 窄转换到 char。

上述所讲的基本数据类型的 3 种转换在 Java 程序中经常发生。例如，当将一个表达式的值赋给一个变量时（称为赋值类型转换上下文），会自动进行“宽转换”和某些特定的“窄转换”，如下面的代码片段。

```
byte b = 2;
short s;
s = ++b;
```

表达式 ++b 是 byte 类型，而 s 是 short 类型，因而期望赋值符“=”右部出现 short 类型。自动执行“宽转换”，将 byte 转换为 short。

赋值时允许的特定的“窄转换”是：若常量表达式（或常数）的类型是 byte、short、char、int，要将值赋给一个变量 V（变量 V 的类型是 byte、short、char）。若常量值处于变量 V 的数据类型范围之内，则编译程序自动进行这种特定的“窄转换”。如 byte v = '\uff00'，则必须要程序员自己明确进行“窄转换”。

另外一种经常发生的类型转换上下文是“数值提升”。当使用算术运算符（+，-，*，/，%），关系运算符（<，<=，>，>=，==，!=），位运算符（&，|，^，~，>>，>>>，<<）及条件运算符（?:），编译程序会按“宽转换”自动进行“数值提升”。例如下面的代码片段。

```
byte b = 10;
long l = 20;
```

对于表达式 b+l，首先将 b “宽转换”成 long，然后按 long 进行“+”运算。结果类型是 long 型。再如：

```
byte b = 10;
char c = '\u0065';
```

对于表达式 b+c，首先将 b 与 c 按“宽转换”自动提升为 int，然后按 int 进行计算，结果类型为 int。

说明：若是 byte、short、char 三者之间进行运算，则首先将它们全部按“宽转换”自动提升为 int，然后按 int 再进行运算。

其他类型转换上下文如方法调用类型转换（如 3.2.4 节方法重载）参见相关章节。

2.2 运算符和表达式

在 Java 中，运算符和表达式是实现数据操作的两个重要的组成部分。运算符是指表达各种运算的符号。表达式是符合一定语法规则的运算符和操作数的序列。

2.2.1 运算符

Java 中表达各种运算的符号称为运算符，运算符的运算对象称为操作数。只需要一个操作数参与运算的运算符称为单目运算符，如+（正号）、-（负号）等。需要两个操作数参与运算的运算符称为双目运算符，如×（乘）、+（加）等。需要 3 个操作数参与运算的运算符称为三目运算符，如?:（条件运算符）。

表 2.3 列出了 Java 中的运算符及其相关的内容。

表 2.3 Java 运算符一览表

优 先 级	运 算 符	描 述	目 数	结 合 性
1(最高)	.	成员运算符	双目	从左向右
	[]	数组下标运算符	双目	
	()	圆括号		
	表达式++	后自增 1 运算符	单目	
	表达式--	后自减 1 运算符	单目	
2	++表达式	前自增 1 运算符	单目	从右向左
	--表达式	前自减 1 运算符	单目	
	!	逻辑非	单目	
	~	按位求反	单目	
	+	正号	单目	
	-	负号	单目	
	(类型)	强制类型转换	单目	
3	new (类型)	内存分配运算符	单目	
4	*	乘	双目	从左向右
	/	实数除或取整		
	%	取余数		
5	+, -	加、减运算符		
6	>>	保留符号的右移		
	>>>	不保留符号右移		
	<<	左移		
7	>, >=, <, <=	大于、大于小于、小于、小于等于、实例运算符		
8	==, !=	相等、不相等		
9	&	位与		
10	^	位异或		
11		位或		
12	&&	逻辑与		
13		逻辑或		
14	?:	条件运算符	三目	从右向左
15 (最低)	=, +=, -=, *=, /=, %=, ^=, &=, =, <<=, >>=, >>>=	赋值运算符	双目	从右向左

1. 算术运算符

算术运算符用于处理整型、浮点型、字符型的数据，进行算术运算。

Java 对 “+” 作了重载（Java 中唯一重载的符号），如 “abc” + var_a，其中变量 “var_a” 可为任何 Java 类型。如 “abc” + 12.6，结果是 “abc12.6”。又如 12.6 + “abc”，结果是 “12.6abc”。又如：int a = 10, b = 11，则 “a + b = ” + a + b 值是 “a + b = 1011”。而 “a + b = ” + (a + b) 值是 “a + b = 21”。

“/” 用于整型表示取整，如 7/2 结果为 3。“/” 用于 float、double 表示实数相除，如 7.0/2 结果为 3.5。例如下面的语句。

```
int a = 7, b = 2; float c; c = a / b;
```

c 的值仍是 3.0f。因此若要使 a/b 按实数除法进行，可用强制类型转换：c = (float)a / b；即先将 a 的类型转换成 float 类型，然后 “/” 将按实数相除进行。

“%” 用于整型表示取余数。例如，15%2 结果为 1，(-15)%2 结果为 -1，15%(-2) 结果为 1，(-15)%(-2) 结果为 -1。“%” 用于 float、double 表示实数取余，如 15.2%5 = 0.2。

“++” 表示自增，有前自增 ++a 与后自增 a++ 两种，其中 a 必须是一个变量。++a 表示：先将 a 的值增加 1，然后 a 的值（已增加 1）即为整个表达式（++a）的值。a++ 表示：先将 a 的值（未增加）作为整个表达式（a++）的值，然后 a 的值增加 1。

“--” 表示自减，有前自减 --a 与后自减 a-- 两种，其中 a 必须是一个变量。--a 表示：先将 a 的值减少 1，然后 a 的值（已减少 1）即为整个表达式（--a）的值。a-- 表示：先将 a 的值（未减少）作为整个表达式（a--）的值，然后 a 的值减少 1。

在 Java 中没有乘幂运算符。若要进行乘幂运算，如 x^y ，则可用类 Math 的 pow() 方法：Math.pow(x,y)。

2. 关系运算符

关系运算符用于比较两个操作数，运算结果是布尔类型的值 true 或 false。所有关系运算符都是二目运算符。Java 中共有 6 种关系运算符：>（大于）、>=（大于等于）、<（小于）、<=（小于等于）、!=（不等于）、==（等于）。前 4 种优先级相同，且高于后面的两种。如 a == b > c 等效于 a == (b > c)。在 Java 中，任何类型的数据（无论是基本数据类型还是引用类型）都可以通过 == 或 != 来比较是否相等或不等。如布尔类型值 true == false 的运算结果是：false。只有 char、byte、short、int、long、float、double 类型才能用于前 4 种关系运算，这些运算符的优先级与结合方向如表 2.3 所示。

3. 逻辑运算符

布尔逻辑运算符用于将多个关系表达式或 true、false 组成一个逻辑表达式。Java 中的逻辑运算符：&（与）、|（或）、!（非）、^（异或）、&&（短路与）、||（短路或）。

a && b：只有 a 与 b 都为 true，结果才为 true；有一个为 false，结果为 false。

a || b：只有 a 与 b 都为 false，结果才为 false；有一个为 true，结果为 true。

!a：与 a 的值相反。

Java 中逻辑表达式进行所谓的“短路”计算。例如，计算 a || b && c 时，若 a 的值为 true，则右边 b && c 就不需进行计算，最后结果一定是 true。只有当 a 为 false 时，右边 b && c 才有执行的机会。当进一步计算 b && c 时，仍是短路计算。当 b 是 false 时，c 不用计算。只

有当 b 为 true 时，c 才有执行的机会。Java 编译程序按短路计算方式来生成目标代码。

这 3 个运算符的优先级与结合方向如表 2.3 所示。

4. 位运算符

位运算符是对操作数按其在计算机内部的二进制表示按位进行操作。参与运算的操作数只能是 int、long 类型，其他类型的数据要参与位运算要转换成这两种类型。Java 中共有 7 种位运算符：

~（按位求反）、&（与）、|（或）、^（异或）、>>（保留符号的右移）、>>>（不保留符号的右移）、<<（左移）。

● 按位求反运算符~

~是单目运算符，对操作数的二进制数据的每一个二进制位都取反，即 1 变成 0，而 0 变成 1。例如，~10010011 结果是 01101100。

● 与运算符&

参与运算的两个操作数，相应的二进制数位进行与运算，即 $0 \& 0 = 0$ ， $0 \& 1 = 0$ ， $1 \& 0 = 0$ ， $1 \& 1 = 1$ ，也即 $x \& 0 = 0$ ， $x \& 1 = x$ 。其中，x 是 0 或 1。例如，a = 11001011，则 a = a & 11111100 的结果是 a 的值变为 11001000，即将数 a 的前六位不变，最后两位清零。保持不变的位，要用 1 去进行&运算，而要清零的位，要用 0 去进行“&”运算。例如，a = 11001010，b = a & 00000011，则 b 的值是 00000010，即取出变量 a 中的最后两位二进制位。

● 或运算符|

参与运算的两个操作数，相应的二进制数位进行或运算，即 $0 | 0 = 0$ ， $0 | 1 = 1$ ， $1 | 0 = 1$ ， $1 | 1 = 1$ ，也即 $x | 0 = x$ ， $x | 1 = 1$ 。其中，x 是 0 或 1。例如，a = 11001000，则 a = a | 00000011 的结果是 a 的值变为 11001011，即将数 a 的前六位不变，最后两位位置 1。保持不变的位，要用 0 去进行“|”运算，而要置 1 的位，要用 1 去进行“|”运算。

● 异或运算符^

参与运算的两个操作数，相应的二进制数位进行异或运算，即 $0 \wedge 0 = 0$ ， $1 \wedge 0 = 1$ ， $0 \wedge 1 = 1$ ， $1 \wedge 1 = 0$ 。即不相同是 1，相同是 0。例如，a = 10100011，则 a = a ^ 10100011 的结果是 00000000。

● 保留符号位的右移运算符>>

将一个操作数的各个二进制位全部向右移若干位，左边空出的位全部用最高位的符号位来填充。例如，a = 10100011，则 a = a >> 2 的结果是 a 的值为 11101000。

向右移一位相当于整除 2，用右移实现除法运算速度要比通常的除法运算速度要快。

设 a 是 32 位的 int 类型变量，如 a = 0x7fffffff，则 b = a >> 31 的结果是 b 的值，为 0x00000000。但 b = a >> 34 的结果是 b 的值并不是 0x00000000，与 b = a >> 2 相同。即 Java 对 32 位整型数的右移，自动先对所要移动的位的个数进行除以 32 取余数的运算，然后再进行右移。即若 a 是 int 型变量，则 a >> n 就是 a >> (n % 32)。同样，若 a 是 64 位的长整数类型，Java 自动先对移动的位的个数进行除以 64 取余数的运算，然后再进行移位，即 a >> n 就是 a >> (n % 64)。例如，a = 0xffffffff77777777L，则 b = a >> 66 与 b = a >> 2 相同。另外，a >> (-2)等价于 a >> (-2+32)，即 a >> 30。

● 不保留符号位的右移运算符>>>

与>>不同的是，右移后左边空出的位用 0 填充。同样在移位之前，自动先对所要移动的位的个数进行除以 32（或 64）取余数的运算，然后再进行右移。即若 a 是 int 型，则 a >>> n 就是 a >>> (n % 32)；若 a 是 long 型，则 a >>> n 就是 a >>> (n % 64)。

● 左移运算符<<

将一个操作数的所有二进制位向左移若干位，右边空出的位填 0。同样，在移位之前，自动先对所要移动的位的个数进行除以 32（或 64）取余数的运算，然后再进行左移。即若 a 是 int 型，则 $a < n$ 就是 $a << (n \% 32)$ ；若 a 是 long 型，则 $a < n$ 就是 $a << (n \% 64)$ 。

在不产生溢出的情况下，左移一位相当于乘以 2，用左移实现乘法运算的速度比通常的乘法运算速度要快。

5. 赋值运算符

● 赋值运算符=

在 Java 中，赋值运算符=是一个双目运算符，结合方向从右向左。它用于将赋值符右边的操作数的值赋给左边的变量，且这个值是整个赋值运算表达式的值。若赋值运算符两边的类型不一致，且右边操作数类型不能自动转换到左边操作数的类型，则需要强制类型转换。例如下面的语句。

```
float f = 2.6f;
int i = f;                                // 出错，因为 float 不能自动转换成 int
```

故应改为：

```
int i = (int) f;                          // 强制类型转换，此时 i 的值是 2
```

● 复合赋值运算符

在 Java 中规定了 11 种复合赋值运算符，如表 2.4 所示。

表 2.4 复合赋值运算符

运 算 符	用 法 举 例	等效表达式
+=	op1 += op2	op1 = (T)(op1 + op2), T 是 op1 类型
-=	op1 -= op2	op1 = (T)(op1 - op2)
*=	op1 *= op2	op1 = (T)(op1 * op2)
/=	op1 /= op2	op1 = (T)(op1 / op2)
%=	op1 %= op2	op1 = (T)(op1 % op2)
&=	op1 &= op2	op1 = (T)(op1 & op2)
=	op1 = op2	op1 = (T)(op1 op2)
^=	op1 ^= op2	op1 = (T)(op1 ^ op2)
>>=	op1 >>= op2	op1 = (T)(op1 >> op2)
<<=	op1 <<= op2	op1 = (T)(op1 << op2)
>>>=	op1 >>>= op2	op1 = (T)(op1 >>> op2)

各种赋值运算符的优先级均相同且是右结合的。赋值运算符的优先级最低，如表 2.3 所示。例如，`int x = 1; x += 2.0f;` 等价于：`x = (int)(x+2.0f)`。

6. 条件位运算符

条件运算符?:是三目运算符，其格式是：

```
e1?e2:e3
```

其中 e1 是一个布尔表达式，若 e1 的值是 true，则计算表达式 e2 的值，且该值即为整个条件运算表达式的值，否则计算表达式 e3 的值，且该值即为整个条件运算表达式的值。 e2 与 e3 需要

返回相同的或兼容的数据类型，且该类型不能是 `void`。例如，`int a=4,b=8; minValue= a>b? b:a ;`。

2.2.2 表达式

表达式是符合一定语法规则的运算符和操作数的序列。一个常量、变量也认为是一个表达式，该常量或变量的值即为整个表达式的值。一个合法的 `Java` 表达式经过计算后，应该有一个确定的值和类型。唯一的例外是方法调用时，该方法的返回值类型被定义为 `void`。通常，不同的运算符构成不同的表达式。例如，关系运算符 `>`、`>=` 等构成关系表达式，关系表达式的值只能取 `true` 或 `false`，其类型为 `boolean` 型。

2.3 流程控制

`Java` 使用了 `C` 语言的所有流程控制语句，如分支语句、循环语句、流程跳转语句等。它们用于控制程序的运行流程。

2.3.1 分支语句

分支语句使部分程序代码在满足特定条件下才会被执行。`Java` 语言支持两种分支语句：`if ... else` 语句和 `switch` 语句。

1. `if ... else` 语句

`if` 语句又称为条件语句，其语法格式为：

```
if (<布尔表达式>) <语句 1>; [else <语句 2>;]
```

其中，<语句 1>及<语句 2>可以是任何语句，包括复合语句，`else` 部分可以有，也可以没有。

`if` 语句的语义是：首先计算<布尔表达式>的值，若值是 `true`，则执行<语句 1>，当<语句 1>执行完成，则整个 `if` 语句就执行结束了；当<布尔表达式>的值是 `false` 时，执行 `else` 部分的<语句 2>，当<语句 2>执行完成，整个 `if` 语句就执行结束了。由于 `if` 语句中的<语句 1>或<语句 2>可以是任何语句，则当<语句 1>或<语句 2> 是另一个 `if` 语句时，就产生了 `if` 语句的嵌套，例如下面的代码片段。

```
if (a>1)
    if (b>10)
        System.out.println(a+b);
    else
        System.out.println(a-b);
```

// 此处的 `else` 与哪一个 `if` 相配？

这个嵌套的 `if` 语句产生了二义性，`else` 与哪一个 `if` 相配呢？`Java` 语言规定：`else` 与最近的没有配上 `else` 的 `if` 相配，故上述的 `else` 与第二个 `if` 相配。若要使该 `else` 与第一个 `if` 相配，则代码如下。

```
if (a>1){
    if (b>10)
        System.out.println(a+b);
}
else
```

// 加上一对 {}，形成一条复合语句

// 此处的 `else` 与第一个 `if` 相配

System.out.println(a-b);

【例 2.1】 设计一个 Java 程序，判断某一年份是否是闰年。

LeapYear.java

```
public class LeapYear {
    public static void main(String[] args){
        // args[0]表示命令行的第一个参数并把它由字符串转换为整形
        int year = Integer.parseInt(args[0]);
        int leap;                                // 1 表示闰年，0 表示不是闰年
        if(year % 4 == 0){                        // 判断能否被 4 整除
            if (year % 100 == 0){
                if (year % 400 == 0)
                    leap = 1;
                else leap = 0;
            }
            else
                leap = 1;                        // 是闰年
        }
        else leap = 0;
        if (leap == 1)
            System.out.println(year + "年是闰年");
        else
            System.out.println(year + "年不是闰年");
    }
}
```

右击“LeapYear.java”，选择“Run As”→“Run Configurations”命令，如图 2.1 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_02”，在“Main class”栏中选择“LeapYear”，选择“Arguments”标签页，在“Program arguments”栏中输入“1984”，然后单击“Run”按钮，运行程序。

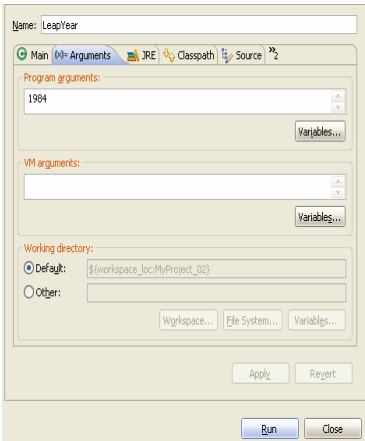


图 2.1 计算闰年

程序运行结果：

1984年是闰年

按照上面同样的方式，在“Program arguments”栏中输入“2009”，然后单击“Run”按钮，运行程序。

程序运行结果：

2009年不是闰年

2. switch 语句

当程序有多个分支（通常超过两个）时，使用 switch 语句比使用多个嵌套的 if 语句更简明些。switch 语句的语法格式如下。

```
switch(<表达式>) {  
    case <常量表达式 1>: [<语句 1>]  
    case <常量表达式 2>: [<语句 2>]  
    ...  
    case <常量表达式 n>: [<语句 n>]  
    [default:<语句 n+1>]  
}
```

其中，<表达式>的最终值的类型必须是 int 型或者是能自动转换成 int 型的类型，如 byte、short、char、int，否则，必须进行强制类型转换。<语句 1>~<语句 n+1>可以是任何语句，也可以缺省不写。<常量表达式 1>~<常量表达式 n>在编译时就可以计算出它们的值，并将计算出的常量值作为相应语句的一个标号。由于常量值兼作标号，故显然要求不能有两个或以上的常量值相同。

switch 语句的语义是：

首先计算<表达式>的值，然后判断该值与<常量表达式 1>的值是否相等，若相等，则从<语句 1>开始，一直执行到<语句 n+1>，即它是一直执行到底的。否则，继续判断该值与<常量表达式 2>的值是否相等，若相等，则从<语句 2>开始，一直执行到底。否则，继续判断该值与<常量表达式 3>的值是否相等。若没有一个常量表达式的值与<表达式>的值相等，则从 default 这个标号所代表的语句开始，一直执行到底。

2.3.2 循环语句

循环语句的作用是反复执行一段代码，直到不满足循环条件为止。循环语句一般应包括如下 4 部分内容。

(1) 初始化部分：用来设置循环的一些初始条件，比如设置循环控制变量的初始值。

(2) 循环条件：这是一个布尔表达式，每一次循环都要对该表达式求值，以判断是否继续循环。

(3) 循环体：这是循环操作的主体内容，可以是一条语句，也可以是多条语句。

(4) 迭代部分：通常属于循环体的一部分，用来改变循环控制变量的值，从而改变循环条件表达式的布尔值。

Java 语言有 3 种循环语句：for 语句、while 语句和 do ... while 语句。for 和 while 语句在执行循环体之前测试循环条件，而 do ... while 语句在执行完循环体之后测试循环条件。这意

意味着 for 和 while 语句可能连一次循环体都未执行，而 do ... while 循环至少执行一次循环体。

1. while语句

while 语句的语法格式是：

```
while(<布尔表达式>) {  
    <语句 1>  
    <语句 2>  
    ...  
    <语句 n>  
}
```

其中，<布尔表达式>是循环继续下去的条件，循环体中的<语句 1>~<语句 n>可以是任意合法的 Java 语句。若循环体中只有一条语句，则循环体的一对花括号{}可以省略不写。

while 语句的语义是：

第 1 步：计算<布尔表达式>的值，若值是 false，则整个 while 语句执行结束，程序将继续执行紧跟在该 while 循环体之后的语句，而循环体中的语句一次都没有得到执行。若值是 true，则转到第 2 步。

第 2 步：依次执行循环体中的<语句 1>~<语句 n>，再转到第 1 步。

注意：循环体或布尔表达式中至少应该有这样的操作，它的执行会改变或影响 while(<布尔表达式>)中<布尔表达式>的值，否则 while 语句就会永远执行下去，不能终止，成为一个死循环。例如：

```
int a = 1, b = 2;  
while( a<b) {  
    b++;  
    a += 2;  
}
```

2. do ... while语句

do...while 语句的语法格式是：

```
do {  
    <语句 1>  
    <语句 2>  
    ...  
    <语句 n>  
} while(<布尔表达式>);
```

其中，<布尔表达式>是循环继续下去的条件，循环体中的<语句 1>~<语句 n>可以是任何合法的 Java 语句。

注意：即使循环体中只有一条语句，循环体的一对花括号({})也不能省略不写。

do ... while 语句的语义是：

第 1 步：依次执行循环体中的<语句 1>~<语句 n>。

第 2 步：计算<布尔表达式>的值，若值是 false，则整个 do ... while 语句执行结束，程

序继续执行紧跟在该 `do ... while` 语句之后的语句。若值是 `true`，则转到第 1 步。

从 `do ... while` 语句的语义中可知，循环体中的语句至少执行一次，即循环体最少执行的次数是 1 次。同 `while` 语句一样，循环体或布尔表达式中至少应该有这样的操作，它的执行会改变或影响 `while (<布尔表达式>)` 中 `<布尔表达式>` 的值，否则，`do ... while` 语句就会永远执行下去，不能终止，成为一个死循环。

【例 2.2】 设计一个 Java 程序，打印出 $1*2*3 \cdots *n$ 之积，变量 n 的初始值在程序中指定。

MultiplyCalculate.java

```
public class MultiplyCalculate {
    public static void main(String[] args){
        long s=1;
        int k = Integer.parseInt(args[0]);           // 把字符串转换为整形
        for (int i = 1;i <k;i++){
            s = s * i ;
        }
        System.out.println("1 * 2 * 3"+ "...* "+k+" = "+ s);    // 打印相乘的结果
    }
}
```

右击 “MultiplyCalculate.java”，选择 “Run As” → “Run Configurations”，如图 2.2 所示，选择 Main 标签页，在 “Project” 栏中选择 “MyProject_02”，在 “Main class” 栏中选择 “MultiplyCalculate”，选择 “Arguments” 标签页，在 “Program arguments” 栏中输入 “10”，然后单击 “Run” 按钮，运行程序。

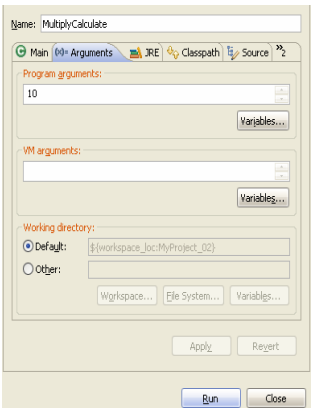


图 2.2 计算整数相乘的结果

程序运行结果：

```
1*2*3...*10 = 3628800
```

3. for 语句

for 语句与 while 语句一样，也是先判断循环条件，再执行循环体。它的语法格式是：

```

for (e1;e2;e3) {
    <语句 1>
    <语句 2>
    ...
    <语句 n>
}

```

其中，e1 是用逗号分隔的 Java 中任意表达式的列表。e1 可以缺省不写，但 e1 后的分号不能省略；e2 是布尔表达式，表示循环继续执行的条件，若该部分缺省，则表示条件永远为真，成为死循环，但 e2 后的分号同样不能省略；e3 同样是用逗号分隔的 Java 中任意表达式的列表，也可以缺省不写。循环体中的<语句 1>~<语句 n>可以是任何合法的 Java 语句。若循环体中只有一条语句，则循环体的一对花括号可以省略不写。

for 语句的语义是：

第 1 步：首先从左到右依次执行 e1 中用逗号分隔的各个表达式，这些表达式仅在此执行一次，以后不再执行，通常用于变量初始化赋值。

第 2 步：计算布尔表达式 e2 的值。若值为 false，则整个 for 循环语句执行结束，程序继续执行紧跟在该 for 语句之后的语句；若值为 true，则依次执行循环体中<语句 1>~<语句 n>。

第 3 步：从左到右依次执行 e3 中用逗号分隔的各个表达式，再转到第 2 步。

【例 2.3】 设计一个 Java 程序，输出所有的水仙花数。所谓水仙花数，是一个 3 位数，其各位数字的立方和等于该数自身，例如， $153 = 1^3 + 5^3 + 3^3$ 。

TestNum.java

```

public class TestNum {
    public static void main(String[] args){
        int i = 0;                // 百位数
        int j = 0;                // 十位数
        int k = 0;                // 个位数
        int n = 0;
        int p = 0;
        for( int m = 100; m <1000; m++){
            i = m /100;           // 得到百位数
            n = m % 100;
            j = n /10;            // 得到十位数
            k = n % 10;           // 得到个位数
            p = i*i*i+j*j*j+k*k*k;
            if (p==m){
                System.out.println(m);    // 打印水仙花数
            }
        }
    }
}

```

程序运行结果：

```
153
370
371
407
```

2.3.3 流程跳转语句

`break`、`continue` 和 `return` 语句用来控制流程的跳转。

(1) `break`: 从 `switch` 语句、循环语句或标号标识的代码块中退出。以下 `while` 循环语句用于计算从 1 加到 100 的值。

```
int a = 1,result = 0;
while (true) {
    result += a++;
    if (a == 101) break;
}
System.out.println(result);
```

(2) `continue`: 跳过本次循环，立即开始新一轮循环。

【例 2.4】 打印 100~200 之间能被 3 整除的数，每 10 个数一行。

TestContinue.java

```
public class TestContinue {
    public static void main(String[] args) {
        int n = 0;
        int i = 0;
        for (n = 100; n <= 200; n++) {
            if (n % 3 != 0)
                continue; // 不能被 3 整除，结束本次循环
            i++;
            System.out.print(n + " ");
            if (i % 10 == 0) { // 每 10 个数一行
                System.out.println();
            }
        }
    }
}
```

程序运行结果:

102	105	108	111	114	117	120	123	126	129
132	135	138	141	144	147	150	153	156	159
162	165	168	171	174	177	180	183	186	189
192	195	198							

(3) `return`: 退出本方法，跳到上层调用方法。如果本方法的返回类型不是 `void`，则需要提供相应的返回值。

2.4 方法与递归

在 Java 语言中，一个方法代表一个具体的逻辑功能，表达方法所属的类的对象所具有的一种行为或一种操作。递归是一种重要的程序设计技术。用递归表达程序设计逻辑十分简明。

2.4.1 方法

Java 中的一个方法类似于其他语言的函数，它代表了一个具体的逻辑功能，表达了它所属的类的对象所具有的一种行为或一种操作。Java 中方法定义的基本格式是：

```
[ 修饰符 1 修饰符 2... ]<返回值类型><方法名>([<形式参数列表>]) {  
    [<方法体>]  
}
```

其中，<返回值类型>可以是任何合法的 Java 的数据类型。若该方法没有返回值，则类型应定义为 void。<方法名>是任何合法的 Java 标识符，Java 保留字不能用做方法名。<形式参数列表>定义该方法要接受的输入值及相应类型，该列表是可以缺省的。<方法体>中是任何合法的 Java 语句，这些 Java 语句共同完成该方法所定义的逻辑功能，该部分也可以缺省。

在下面的 TestMethod 类中，定义了一个方法 simpleMethod。

```
public class TestMethod {  
    int simpleMethod(int x, int y ) {  
        ...  
        return (x<y)?x:y ;  
    }  
}
```

在方法 simpleMethod 中，定义了两个形式参数 x 和 y，返回两者之中的最小值。

2.4.2 参数传递

Java 规定：所有类型的参数传递都是“值传递”。其机制是：在方法调用运行时，首先对实在参数列表从左到右依次计算各个实在参数的值，然后在运行栈中为该方法的所有的形式参数变量分配空间，接着再为该方法体中所有其他局部变量在运行栈中分配空间，最后将已计算出来的各个实在参数的值抄送到相应的形式参数变量空间之中。这一切做完后，方法调用才开始执行方法体中的第一条语句。一旦运行完成，自动从运行栈撤销该方法的所有信息，因而该方法在运行栈中为形式参数及局部变量分配的空间也就自动撤销。

【例 2.5】 打印 100 之间所有的质数。

PrimeNumber.java

```
public class PrimeNumber {  
    boolean isPrime(int n) {  
        for (int i = 2; i <= n / 2; i++)  
            if (n % i == 0)  
                return false;  
        return true;  
    }  
}
```

```

void printPrime(int m) {
    int j = 0;
    for (int i = 2; i <= m; i++){
        if (isPrime(i)){
            j++;
            if(j%10==0){                // 每 10 个质数一行
                System.out.print(i + "\t");
                System.out.println();    // 换行
            }
            else
                System.out.print(i + "\t");
        }
    }
}

public static void main(String[] args) {
    PrimeNumber pn = new PrimeNumber();
    pn.printPrime(100);                // 打印出 100 之内的所有质数
}
}

```

程序运行结果：

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97					

2.4.3 递归

递归是相当重要的一种程序设计技术，用递归表达程序设计逻辑十分简明。递归分为两种：直接递归与间接递归。

直接递归是指方法运行时又调用了自身。间接递归是指方法运行时通过调用其他方法，最终又调用自身，如图 2.3 所示。

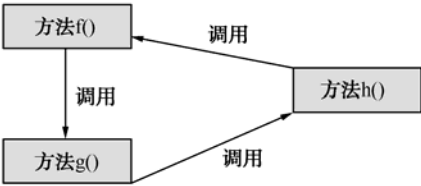


图 2.3 间接递归

【例 2.6】 递归计算 $1+2+\cdots+100$ 之和。

Sum.java

```

public class Sum {

```

```

static int P(int n) {
    return n == 0 ? 0 : n + P(n-1);
}

public static void main(String[] args) {
    System.out.println("1+2+...+100 之和是: " + P(100));
}
}

```

程序运行结果：

```
1+2+...+100 = 5050
```

【例 2.7】 递归计算 Fibonacci 数列的第 10 个值。

Fibonacci.java

```

public class Fibonacci {
    public static void main(String[] args) {
        System.out.println(f(5));
    }

    public static int f(int n){
        if (n == 1 || n == 2){
            return 1;
        }else {
            return f(n-1)+ f(n-2);
        }
    }
}

```

程序运行结果：

```
55
```

2.5 命名规范

在 Java 中，代码的编写需遵循一定的规范。Java 编程规范的主要内容有：

(1) 类名和接口名：首字母大写，如果类名由几个单词组成，采用驼峰标识，即每个单词的首字母大写，其余字母小写，例如，**HelloWorld**。

(2) 方法名和变量名：首字母小写（构造方法除外），如果方法名或变量名由几个单词组成，那么除第一个单词外，其余每一个单词的首字母大写，其他字母小写，例如，**toString()**。

(3) 包名：采用小写形式，例如，**org.innerclasses.anonymous**。

(4) 常量名：采用大写形式，如果常量名由几个单词组成，单词之间以下画线“_”隔开，例如，**final int VALUE_ONE = 9**。

2.6 注释语句

在 Java 源文件的任意位置, 都可以加入注释语句, Java 编译器会忽略程序中的注释语句。Java 语言提供了 3 种形式的注释。

- (1) `// java annotation` —— 从 “`//`” 到本行结束的所有字符均作为注释而被编译器忽略。
- (2) `/*java annotation*/` —— 从 “`/*`” 到 “`*/`” 间的所有字符会被编译器忽略。
- (3) `/** java annotation*/` —— 从 “`/**`” 到 “`*/`” 间的所有字符会被编译器忽略。

Sun 公司为 JDK 开发工具包提供了完整的 JDK 文档说明, JDK 文档中提供了 Java 中的各种技术的详细资料, 以及 JDK 中提供的各种类的用法说明。

对于用户创建的 Java 类, 如何编写这种 html 格式的 javadoc 文档呢? JDK 中提供了一个 `javadoc.exe` 程序, 它能够识别 Java 源文件中符合特定规范的注释语句, 根据这些注释语句自动生成 javadoc 文档。在 Java 源文件中, 只有满足特定的注释才会构成 javadoc 文档。这些规范包括:

- ① 注释以 “`/**`” 开始, 并以 “`*/`” 结束, 里面可以包含普通文本、html 标记和 javadoc 标记。
- ② `javadoc` 命令只处理 Java 源文件中在类声明、接口声明、成员方法声明、成员变量声明及构造方法声明之前的注释, 而忽略位于其他地方的注释。

2.7 综合实例

功能: 求某一个日期对应的是星期几。

用 3 个正数 `year`、`month`、`day` 分别记载一个日期的年、月、日, 计算至今的总天数 `total`。本实例以 1980 年 1 月 1 日 (星期二) 为起始日。

总天数 = 平年累计值 + 闰年累计值 + 当前前几月的累计天数 + 本月天数
求总天数 `total` 的步骤如下。

- (1) `total` 的初值 = 平年累计值 + 闰年累计值。

因为平年有 365 天, 闰年有 366 天, 而 $365 \% 7 = 1$, 所以平年的总天数每年只需累计 1, 闰年累计 2 即可。因此, `total` 的初值为:

$$total = year - 1980 + (year - 1980 + 3) / 4$$

其中, `year-1980` 是 `year` 与 1980 相距的年数, 即平年累计值; $(year-1980+3)/4$ 是 `year` 与 1980 年间相距的闰年数, 即闰年累计值。

当 `year = 1980` 年时为闰年, 当年闰年值不计, 所以 $(year-1980+3) / 4 = 0$; 而 `year` 为 1981~1983 时, 应计 1980 年的闰年值, 所以 $(year-1980+3) / 4 = 1$ 。

- (2) 计算当前前几月的累计天数, 加到 `total` 上。
- (3) 将本月天数加到 `total` 上。

因起始日的前一日为星期一, 故 `week` 的初值为 1, 通过计算下式求得所求日期是星期几。

$$week = (week + total) \% 7$$

程序如下。

CalculateWeekDay.java

```
import java.util.*;
import java.text.*;

public class CalculateWeekDay {
    public static void main(String[] args) {
        Date date = new Date();
        SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd");    // 格式化日期
        System.out.println(f.format(date));
        String str = f.format(date);
        String str1[] = str.split("-");    // 分隔日期字符串
        int year = Integer.parseInt(str1[0]);    // 使用包装类把字符型转换为整型
        int month = Integer.parseInt(str1[1]);
        int day = Integer.parseInt(str1[2]);
        int total, week, i;
        boolean leap = false;
        leap = (year % 400 == 0) | (year % 100 != 0) & (year % 4 == 0); // 判断当年是否是闰年
        week = 1;    // 起始日 1979-12-31 是 monday
        total = year - 1980 + (year - 1980 + 3) / 4;    // 计算 total 的初值
        // 计算当年前几月的累计天数与 total 的初值之和
        for (i = 1; i <= month - 1; i++) {
            switch (i) {    // 判断当前月份
                case 1: case 3: case 5: case 7: case 8: case 10:
                case 12: total = total + 31; break;
                case 4: case 6: case 9:
                case 11: total = total + 30; break;
                case 2:
                    if (leap) total = total + 29;
                    else total = total + 28;
                break;
            }
        }
        total = total + day;    // 将本月天数加到 total 上
        week = (week + total) % 7;
        System.out.print("today " + year + "-" + month + "-" + day + " is ");
        switch (week) {
            case 0: System.out.println("Sunday"); break;
            case 1: System.out.println("Monday"); break;
            case 2: System.out.println("Tuesday"); break;
            case 3: System.out.println("Wednesday"); break;
            case 4: System.out.println("Thursday"); break;
```

```
        case 5: System.out.println("Friday"); break;
        case 6: System.out.println("Saturday"); break;
    }
}
```

程序运行结果:

```
2009-06-30
today 2009-6-30 is Tuesday
```

第 3 章 Java面向对象编程（上）

在传统的结构化程序设计中，数据和对数据的操作是相分离的。Java 是优秀的面向对象程序设计语言，它将数据及对数据的处理操作“封装”到一个类中，用“类”或“接口”这些较高层次的概念来表达应用问题，类（Class）的实例称为对象（Object）。基于对象来进行应用问题的分析、思考、设计及代码编写，就是面向对象程序设计（OOP）。

1. 类和对象

类描述具有相同特性和行为的对象的抽象，对象是类的一个实例。例如，“汽车”就是一个类，它包含了汽车的共同特征（例如：型号、发动机排量、外观尺寸、颜色等），而对于“奥迪 A6”则是汽车这个类的一个具体实例。日常生活中会涉及到各种类，例如学生、学校、房子、蔬菜，等等。

子类是在已有类的基础上修改而形成的类，子类所依托的类称为父类。例如：卡车属于汽车，但卡车又有其特有的特征（如载重量），我们可以定义一个卡车子类，它继承汽车类，同时它增加如载重量特性。这时，汽车是卡车的父类。父类修改后，子类将继承任何对父类所作的修改。

2. 属性和方法

在 Java 中，为了实现常用功能，系统提供了一些类称为基类，用户可根据需要自定义类。在 Java 中，类的特征用属性描述，类的行为用方法描述。方法实际上就是一个函数，函数中的语句组成的程序实现类的行为的功能。例如，可以定义一个圆类，它的属性包括半径、周长、面积、圆心坐标等。它的方法是计算周长、计算面积、画圆等。

3. 事件

一个程序运行时会有许多个对象，这些对象之间总是要发生联系的。对象之间可以通过事件进行通信。事件也是一种对象，称为事件对象。事件对象“封装”了该事件所有必须的有用的信息。如，事件源、事件性质、发生时间、发生位置、其他参数等。当对象 A 要与对象 B 通信时，将所有必要信息包装成一个事件对象，然后将该对象传递给对象 B 的特定的方法，对象 B 通过该方法，接收事件对象，完成对事件对象应该进行处理的动作。

3.1 类的定义及成员变量初始化

类实际上就是一个数据类型，只不过它与 Java 的基本数据类型有些区别。Java 的基本数据类型包括布尔型、字节型、短整型、整型、长整型、字符型、单精度浮点型、双精度浮点型等。而类可以认为是一种复杂的数据类型，它可以包括 Java 的基本数据类型、引用类型，描述的是类的属性，也可以包括方法，描述的是类的行为。

3.1.1 类的定义

基本的 Java 类定义格式是：

```
[类修饰符] class <类名> {  
    <类体>  
}
```

其中，关键字 `class` 表示定义一个类，类修饰符可以用 `public` 或缺省，`<类名>`是 Java 合法的标识符名。按 Java 编码约定，类名的英文单词的第一个字母要大写，若由多个单词组成，则每一个单词的首字母都要大写，`<类体>`可以缺省，`<类体>`由变量定义（称为成员变量）和方法定义（称为成员方法）组成。

类的成员变量的声明格式是：

```
[修饰符] <类型> <变量名> [= 初始值];
```

例如，`private String name = “Tom”;`

类的成员方法的声明格式是：

```
[修饰符]<返回值类型><方法名>(形式参数列表) {  
    ...  
}
```

3.1.2 变量的初始化

成员变量可以是 Java 语言中任何一种数据类型(包括基本类型和引用类型)。在定义成员变量时可以对其进行初始化，如果未对其进行初始化，Java 使用默认的值对其进行初始化，如表 3.1 所示，且作用域为整个类体。

注意：在方法内部(包括 `main()` 方法)定义的变量是局部变量。所有局部变量都是在方法被调用时在栈中分配空间，系统不会自动对它们赋初值，所以必须先给局部变量赋初值之后才能使用。

表 3.1 数据类型的默认初始化值

数据类型	默认值
boolean	false
byte	(byte)0
char	'\u0000'
int	0
short	(short)0
long	0L
float	0.0f
double	0.0d
引用类型	null

3.2 创建对象

对象即类的实例化。在 Java 虚拟机的生命周期中，一个个对象被创建，又一个个被销毁。在对象的生命周期的开始阶段，需要为对象分配内存，并且初始化它的实例变量。当程序不再使用某个对象时，它就会结束生命周期。它的内存可以被 Java 虚拟机的垃圾回收器回收。在 Java 程序中，对象可以被显式地或隐含地创建。创建一个对象就是指构造一个类的实例。用 `new` 语句创建对象是 Java 语言创建对象的最常见的方式。

3.2.1 构造方法

在多数情况下，初始化一个对象的最终步骤是去调用这个对象的构造方法。构造方法的功能是：当创建一个类的对象时，首先用 `new` 语句从堆区中分配该对象的内存空间，然后自动调用类的某一个构造方法，对该对象的内存空间进行初始化，为实例变量赋予合适的初始

值。构造方法的语法规则：

- 方法名必须与类名相同。
- 不要声明返回类型。
- 不能被 `static`、`final`、`synchronized`、`abstract` 和 `native` 修饰。

【例 3.1】 不带形参的构造方法，在构造方法中对其成员变量初始化。

ConstructorInitiate.java

```
public class ConstructorInitiate {
    boolean t;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    char c;
    ConstructorInitiate() {
        t = true;                // 成员变量的初始化
        b = 12;
        s = 99;
        i = 49;
        l = 120;
        f = 3.14f;
        d = 3.1415;
        c = 'h';
    }
    public static void main(String[] args) {
        ConstructorInitiate initiate;                // 定义引用变量
        initiate = new ConstructorInitiate();
        System.out.println(initiate.t);
        System.out.println(initiate.b);
        System.out.println(initiate.s);
        System.out.println(initiate.i);
        System.out.println(initiate.l);
        System.out.println(initiate.f);
        System.out.println(initiate.d);
        System.out.println(initiate.c);
    }
}
```

Java 虚拟机在执行下面语句时，将在栈内存中生成一个引用变量 `initiate`，由于 `initiate` 是一个局部变量，且这时还没有创建任何对象，所以 `initiate` 还没有被赋值。

`ConstructorInitiate initiate;`

执行下面的语句后，将在堆内存中创建一个 `ConstructorInitiate` 对象，并对其成员变量进行初始化，并把对象的引用赋给 `initiate`。

`initiate = new ConstructorInitiate();`

这时，`initiate` 指向 `ConstructorInitiate` 对象，其内存布局如图 3.1 所示。

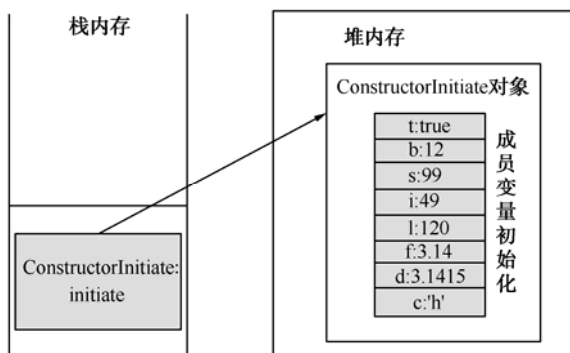


图 3.1 对象的内存布局

程序运行结果：

```
true
12
99
49
120
3.14
3.1415
h
```

既然在创建对象时，会对类的成员变量进行初始化，那么初始化的时机和顺序又是如何？下面的程序解释了初始化时机和顺序。

3.2.2 默认构造方法

Java 语言规定，每个类都必须至少定义一个构造方法。若一个类没有定义构造方法，则编译程序提供一个构造方法。无参数的构造方法称为默认构造方法。编译程序自动提供的构造方法就是一个默认构造方法。格式如下：

```
<与类相同的访问控制符><方法名>(){
    super();                      // 自动调用父类的默认构造方法
}
```

说明：若类是 `public`，自动生成的默认构造方法也是 `public`。若是 `private`(如内部类)，则该默认构造方法也是 `private`。由于方法体中有 `super();`语句，因而要求该类的父类必须要定义一个默认构造方法，若没有定义，则编译报错。这种错误在 Java 初学者中是经常发生的。

若一个类中已定义构造方法(不论有无参数)，编译程序将不再自动提供上述格式的默认构造方法了。因此，当给类定义构造方法时，建议同时定义一个无参构造方法，即默认构造方法，以避免前面所述的那种编译错误。例如下面的代码片段：

```
public class Dog1 {                      // 编译程序自动提供默认方法
    System.out.println("run fast");
}
public class Dog2{                       // 该类缺少默认构造方法。建议程序员自定义一个
    public Dog2(String s) {
        System.out.println("run fast");
    }
}
```

```
public class Dog3 {                                // 自己定义了默认构造方法
    public Dog3(){
        System.out.println(" run fast");
    }
}
```

可以调用 Dog1 类的默认构造方法来创建 Dog1 对象。

```
Dog1 d1 = new Dog1();                             // 合法
```

Dog2 类没有默认的构造方法，因此以下语句会导致编译错误。

```
Dog2 d2 = new Dog2();                             // 编译出错
```

Dog3 类显式定义了默认的构造方法，因此以下语句是合法的。

```
Dog1 d3 = new Dog3();                             // 合法
```

3.2.3 构造方法重载

当通过 new 语句创建一个对象时，在不同条件下，对象可能有不同的初始化行为。既然构造器的名字已经由类名决定，就只能有一个构造器名。那么要想用多种方式创建一个对象该怎么办呢？假设要创建一个类，既可以用标准方式进行初始化，也可以从文件里读取信息来初始化。这就需要两个构造器：一个默认构造器；另一个取字符串作为形式参数。由于都是构造器，所以它们必须用到相同的名字，即类名。为了让方法名相同而形参不同的构造器同时存在，必须用到方法重载。方法的重载是指一个类中可以定义有相同的名字，但参数不同的多个方法。各方法之间的参数个数、参数类型、排列顺序不同即可构成重载，有普通方法重载和构造方法重载。

【例 3.2】 重载构造方法 Student()，并根据实在参数调用相应的构造方法。

ConstructorOverLoad.java

```
class Student {
    private String name = "Lucy";
    private int age = 18;
    Student() {                                    // 无参的构造方法
        System.out.println("invoke no parameter construcor method");
        System.out.println("name is " + name + ",age is " + age);
    }
    Student(String n) {                            // 带有 1 个参数的构造方法
        name = n;
        System.out.println("invoke construcor method with one parameter");
        System.out.println("name is " + name + ",age is " + age);
    }
    Student(String n, int i) {                    // 带有 2 个参数的构造方法
        name = n; age = i;
        System.out.println("invoke construcor method with two parameters");
        System.out.println("name is " + name + ",age is " + age);
    }
}

public class ConstructorOverLoad {
    public static void main(String[] args) {
        new ConstructorOverLoad();
    }
}
```

```

        new Student();
        new Student("Tom");
        new Student("Jack", 25);
    }
}

```

程序运行结果：

```

invoke no parameter construcor method
name is Lucy,age is 18
invoke construcor method with one parameter
name is Tom,age is 18
invoke construcor method with two parameters
name is Jack,age is 25

```

3.2.4 普通方法重载

同构造方法一样，普通方法也可重载。下面的代码片段重载了普通方法 f()，如下所示。

```

void f(int x) {
    ...
}
void f(int x, int y) {
    ...
}
void f(int x, String s) {
    ...
}

```

注意：只有返回值类型不同是不行的，例如下面的语句。

```

int f(int x,int y){...}
double f(int x,int y){...}

```

方法重载是编译时多态性的表现，Java 编译程序会根据方法调用的实在参数来决定使用哪一个方法。

【例 3.3】 测试方法重载，程序重载了 sort()方法，根据所传递的参数个数来调用相应的 sort()方，其 sort()方法用来对 2 个整数或 3 个整数按从小到大排序。

TestSort.java

```

class SortDemo {
    int max, midst, mix;
    SortDemo() {
        max = -1; midst = -1; mix = -1;
    }
    void sort(int i, int j) {                // 两个数排序
        int s; max = i; mix = j;
        if (max < mix) {
            s = max; max = mix; mix = s;
        }
    }
    void sort(int i, int j, int k) {        // 3 个数排序
        int s; max = i; midst = j; mix = k;

```



```

        if (max < midst) {                                // 第一个数和第二个数比较
            s = max;    max = midst;    midst = s;
        }
        if (max < mix) {                                  // 第一个数和第三个数比较
            s = max;    max = mix;    mix = s;
        }
        if (midst < mix) {                                // 第二个数和第三个数比较
            s = midst;    midst = mix;    mix = s;
        }
    }
}

public class TestSort {
    public static void main(String args[]) {
        SortDemo sd = new SortDemo();
        sd.sort(30, 60);
        System.out.println("两个数从大到小为: " + sd.max + "," + sd.mix);
        sd.sort(20, 80, 50);
        System.out.println("三个数从大到小为: " + sd.max + "," + sd.midst + "," + sd.mix);
    }
}

```

程序运行结果:

```

两个数从大到小为: 60,30
三个数从大到小为: 80,50,20

```

3.3 this引用

this 是 Java 的关键字，用于表示对象自身的引用值。当在类中使用实例变量 **x** 或实例方法 **f()** 时，本质上都是 **this.x** 或 **this.f()**。在不混淆的情况下（如：没有名字隐藏），**this.x** 可简写成 **x**，**this.f()** 可简写成 **f()**。

当类中有两个同名变量，一个属于类的成员变量，另一个属于某个特定的方法(方法中的局部变量)，可使用 **this** 区分成员变量和局部变量。使用 **this** 还可以简化构造方法的调用。一个类的实例的成员方法在内存中只有一份备份，尽管在内存中可能有多个对象，而数据成员在类的每个对象所在内存中都存在一份备份。**this** 引用允许相同的实例方法为不同的对象工作，每当调用一个实例方法时，引用该实例方法的特定类对象的 **this** 将自动传入实例方法代码中，方法的代码接着会与 **this** 所代表的对象的特定数据建立关联。在类的 **static** 方法中，是不能使用 **this** 的。这是因为类方法是一直存在，随时可用的，而此时可能该类一个对象都没有创建，自然 **this** 也就不存在。**this** 通常用在构造方法实例变量初始化表达式、实例初始化代码块或实例方法中。除此之外出现 **this**，编译报错。另外，认为 **this** 是当前对象的一个数据成员，这种理解是错误的。

【例 3.4】 使用 **this** 引用来调用类 **A** 方法 **f1()**。

TestThis.java

```

class A{
    String name;

```

```

public A(String str){
    name = str;
}
public void f1(){
    System.out.println("f1() of name "+ name+" is invoked!");
}
public void f2(){
    A a2 = new A("a2");
    this.f1();           // 使用 this 引用调用 f1()方法
    a2.f1();
}
}
public class TestThis {
    public static void main(String[] args){
        A a1 = new A("a1");
        a1.f2();
    }
}

```

程序运行结果：

```

f1() of name a1 is invoked!
f1() of name a2 is invoked!

```

3.4 静态成员

`static` 修饰符可以用来修饰类的成员变量、成员方法和代码块。

- 用 `static` 修饰的成员变量表示静态变量。可以直接通过类名来访问，或通过对象引用来访问。
- 用 `static` 修饰的成员方法表示静态方法，可以直接通过类名来访问，或通过对象引用来访问。
- 用 `static` 修饰的程序代码块表示静态代码块，当 Java 虚拟机加载类时，就会执行该代码块。

3.4.1 `static`关键字

类的成员变量有两种：一种是被 `static` 修饰的变量，叫类变量或静态变量；另一种是没有被 `static` 修饰的变量，叫实例变量。

静态变量和实例变量的区别如下：

- (1) 静态变量在内存中占用一份备份，运行时 Java 虚拟机只为静态变量分配一次内存，在加载类的过程中完成静态变量的内存空间分配。可以直接通过类名访问静态变量。
- (2) 对于实例变量，每创建一个实例，就会为实例变量分配一次内存。实例变量可以在内存中有多份备份，互不影响。。

同成员变量一样，成员方法分为静态方法和实例方法。用 `static` 修饰的方法是静态方法或类方法。访问静态方法不需要创建类的实例，可以通过类名来访问。若已创建了对象，也可通过对象引用来访问。

【例 3.5】 访问静态变量和静态方法。

TestStaticMember.java

```
class Citizen{
    private static String country = "china";           // 静态变量
    private String name = "Tom";
    public static void f1(){
        System.out.println(country);                 // 访问静态变量
        // System.out.println(name);                 // 不可直接访问非静态成员，只能间接访问
    }
    public void f2(){                                 // 可访问静态成员
        f1();                                         // 或 Citizen.f1(); 或 this.f1();
        System.out.println(country);
        System.out.println(name);
    }
}

public class TestStaticMember{
    public static void main(String[] args){
        new TestStaticMethod();
        Citizen.f1();                                // 直接通过类名访问 f1()
        Citizen citizen = new Citizen();
        citizen.f2();
        citizen.f1();                                // 可以通过引用访问 f1()
    }
}
```

程序运行结果：

```
china
china
china
Tom
china
```

在使用类的静态方法时，需要注意以下三点：

(1) 在静态方法里只能直接访问类中其他的静态成员(包括变量和方法)，而不能直接访问类中的非静态成员。这是因为，对于非静态的方法和变量，需要先创建类的实例后才可使用。

```
int i;
public static void f() {
    i = 1;                                           // 错误，不能直接访问非静态变量
    method();                                       // 错误，不能直接访问非静态方法
}
public void method() {
    ...
}
```

(2) 静态方法不能以任何方式引用 **this** 和 **super** 关键字。因为静态方法在使用前是不需要创建任何对象的，当静态方法被调用时，**this** 所引用的对象根本就没有产生。

(3) 子类只能继承、重载、隐藏父类的静态方法，子类不能重写父类的静态方法，也不能把父类不是静态的方法重写成静态的方法。

3.4.2 main()方法

如果一个类要被 Java 解释器直接装载运行，这个类中必须有 main()方法，由于 Java 虚拟机需要调用类的 main()方法，所以该方法的访问权限必须是 public，又因为 Java 虚拟机在执行 main()方法时不必创建对象，所以该方法必须是 static 的，该方法接受一个 String 类型的数组参数，该数组中保存从命令行给 main()方法传递的参数，main()方法执行结束后不返回任何类型，所以该方法的返回类型是 void，因此 main()方法的修饰符是 public static void 的。正因为 main()是静态的，所以在 main()方法中不能直接访问实例变量和实例方法。所有在方法内部(包括 main()方法)定义的变量都是局部变量。

main()方法带有一个 String 类型的数组，该数组中保存执行 Java 命令时传递给所运行的类的参数。

【例 3.6】 从命令行给 main()方法传递参数进行四则运算。

MainOfCommand.java

```
public class MainOfCommand {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);           // 获取第一个命令行参数
        int j = Integer.parseInt(args[2]);           // 获取第三个命令行参数
        int num = 0;                                  // 运算结果
        if (args[1].equals("+")) {
            num = i + j;
            System.out.println(i + " + " + j + " = " + num);
        } else if (args[1].equals("-")) {
            num = i - j;
            System.out.println(i + " - " + j + " = " + num);
        } else if (args[1].equals("x")) {
            num = i * j;
            System.out.println(i + " x " + j + " = " + num);
        } else if (args[1].equals("/")) {
            num = i / j;
            System.out.println(i + " / " + j + " = " + num);
        }
    }
}
```

右击“MainOfCommand.java”，选择“Run As”→“Run Configurations”，如图 3.2 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_03”，在“Main class”栏中选择“MainOfCommand”，选择“Arguments”标签页，在“Program arguments”栏中输入“10 + 2”，然后单击“Run”按钮，运行程序。



图 3.2 四则运算

程序运行结果：

```
10 + 2 = 12
```

按上述同样的过程，在“Program arguments”栏中输入“40 / 5”，然后单击“Run”按钮，运行程序。

程序运行结果：

```
40 / 5 = 8
```

3.4.3 类的初始化

类中可以包含静态代码块，它不存在于任何方法体中。在 Java 虚拟机加载类时会执行这些静态代码块。如果类中包含多个静态代码块，那么 Java 虚拟机将按照它们在类中出现的顺序依次执行它们，每个静态代码块只会被执行一次。示例定义了两个静态代码块和一个构造方法。Java 虚拟机先执行静态代码块，后才调用构造方法。

【例 3.7】 测试静态代码块与构造方法的顺序。

StaticCode.java

```
public class StaticCode {
    static int i=1;
    public StaticCode(){
        System.out.println("initialize construct method"+i++);
    }
    static {                                // 第一个静态代码块
        System.out.println("initialize the first static block"+i++);
    }
    static {                                // 第二个静态代码块
        System.out.println("initialize the second static block"+i++);
    }
    public static void main(String[] args){
        new StaticCode();
        new StaticCode();
    }
}
```

```
}
```

程序运行结果:

```
initialize the first static block1
initialize the second static block2
initialize construct method3
initialize construct method4
```

结果分析: Java 虚拟机首先装载类 `StaticCode`, 然后按如下次序运行:

- (1) 执行 `static int i = 1;` 语句对 `static` 变量 `i` 进行初始化;
- (2) 执行第一个静态代码块;
- (3) 执行第二个静态代码块;
- (4) 调用 `main()` 方法;
- (5) 执行第一个 `new StaticCode();` 语句, 调用构造方法;
- (6) 执行第二个 `new StaticCode();` 语句, 调用构造方法;

注意: `static` 型成员变量与 `static` 代码块是按照它们在类中定义的先后顺序依次执行的。

3.5 package与import语句

为便于管理大型软件系统中数目众多的类, 解决命名冲突问题, Java 引入了包(package)机制, 提供类的多重命名空间。一个包就相当于一个文件夹, 包中的类相当于文件夹下的文件。包与包中的类之间的关系, 相当于文件夹与文件夹中的文件之间的关系。

package 语句的格式为:

```
package pkg1[.pkg2[.pkg2...]];
```

package 语句作为 Java 源文件的第一条语句, 指明该文件中定义的类所在的包(若省略, 则指定为无名包)。Java 编译器把包对应于文件系统的目录管理, package 语句中, 用 “.” 来指明包(目录)的层次, 例如下面的语句:

```
package org.MyProject;
```

表示该文件中所有的类位于 `.\org\MyProject` 目录下。

如果一个类访问了位于另一个包中的类, 那么前者必须通过 `import` 语句把这个类引入。例如, 要使用 `java.awt` 包中的 `AWTEvent` 类, 可通过以下语句导入该类:

```
import java.awt.AWTEvent;
```

还可以导入一个包中的所有类, 例如, 下面的语句用来导入 `java.awt` 包的所有类,

```
import java.awt.*;
```

`import` 语句能够引入一个包中的直接类, 但不能自动引入该包的子包, 必须显式声明导入子包, 例如下面的语句:

```
import java.awt.*;
```

```
import java.awt.event.*;
```

`java.lang` 包无需显式导入, 它总是被编译器自动调入, 但是使用 `java.lang` 包下的子包时仍需显式导入, 例如下面的语句:

```
import java.lang.annotation;
```

在某些情况下, 一个源程序需要导入多个包, 这些包之间有相同的类名, 例如下面的语句:

```
import java.util.*;
import java.sql.*;
```

在这两个包中都有一个重名的日期类 `Date`，编译器无法确定程序使用的是哪一个 `Date` 类，解决的方法是增加一个特定的 `import` 语句。

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

【例 3.8】 对文件打包。

`org.mypackage2.TestPackage.java` 使用了 `Date` 类，并导入了另一个包 `org.myPackage1` 中的类 `Count.java`。为了展示包名的生成，这里暂且改在命令提示符下编译和运行程序。

Count.java

```
package org.mypackage1;
import java.util.Date;
public class Count {
    public void m1(int i, int j) {
        System.out.println(i + j);
    }
    public int m2(int i, int j) {
        return i > j ? i : j;
    }
}
```

TestPackage.java

```
package org.mypackage2;
import java.util.*;
import java.sql.*;
import java.util.Date;
import org.mypackage1.Count;
public class TestPackage {
    public static void main(String[] args){
        Date date1= new Date();           // 等同于 Date date1 = new java.util.Date();
        Date date2 = new java.sql.Date(0);
        System.out.println(date1);
        System.out.println(date2);
        Count count = new Count();
        count.m1(3, 5);
        System.out.println(count.m2(3, 5));
    }
}
```

假如 `Count.java` 与 `TestPackage.java` 存放在 D 盘根目录下，首先进入命令提示符窗口，由于 `Count.java` 位于 `org.mypackage1` 下，输入以下命令：

```
javac -d . Count.java
```

(-d)选项表示生成与包名相对应的目录结构，Java 虚拟机自动在 D 盘创建了 `org\mypackage1` 目录，点(.)代表当前目录，也就是将编译生成的 `Count.class` 二进制文件存放在所创建的目录 `org\mypackage1` 下。

再次输入以下命令：

```
javac -d . TestPackage.java
```

此时，在 D 盘的 org\mypackage2 目录下又生成一个 TestPackage.class 二进制文件。执行二进制文件，输入以下命令：

```
java org.mypackage2.TestPackage
```

如图 3.3 所示。

```
C:\Documents and Settings\tping>D:

D:\>javac -d . Count.java

D:\>javac -d . TestPackage.java

D:\>java org.mypackage2.TestPackage
Sat Nov 22 08:30:38 CST 2008
1970-01-01
8
5
```

图 3.3 包的使用

程序运行结果：

```
Fri Nov 21 17:02:34 CST 2008
1970-01-01
8
5
```

JDK 中有一个 jar 命令，存放在 JDK 安装目录下的 bin 目录中，可以用来对大量的类(.class 文件)进行压缩，然后存为 .jar 文件。现在就可以用 jar 命令把在 D 盘生成的 org 目录及文件压缩。输入以下命令：

```
jar cvf org.jar org
```

现对其命令进行简单解释。

- c 创建新的 JAR 文件包，
- v 生成详细报告并显示到标准输出，
- f 指定 JAR 文件名，通常这个参数是必须的，

“org”表示将被压缩的目录名，“org.jar”表示被压缩后所生成的文件名。

执行了 jar 命令后，将在窗口中显示出打包过程的详细信息，如图 3.4 所示。

```
D:\>jar cvf org.jar org
标明清单(manifest)
增加: org/<读入= 0> <写出= 0><存储了 0%>
增加: org/mypackage1/<读入= 0> <写出= 0><存储了 0%>
增加: org/mypackage1/Count.class<读入= 467> <写出= 326><压缩了 30%>
增加: org/mypackage2/<读入= 0> <写出= 0><存储了 0%>
增加: org/mypackage2/TestPackage.class<读入= 653> <写出= 431><压缩了 33%>
```

图 3.4 文件打包

为了验证上述打包是否完全和正确，可输入以下命令：

```
jar tf org.jar
```

-t 列出 JAR 文件包的内容列表，如图 3.5 所示。


```
D:\>jar tf org.jar
META-INF/
META-INF/MANIFEST.MF
org/
org/mypackage1/
org/mypackage1/Count.class
org/mypackage2/
org/mypackage2/TestPackage.class
```

图 3.5 目录列表

从 JDK5.0 开始, import 语句不仅可以导入类, 还增加了导入静态方法和静态变量的功能。Math 类中的所有方法都是静态方法, 如果对 Math 类使用静态导入, 就可以采用更加自然的方式使用算术方法。例如下面的代码片段中的第一句比第二句要清晰的多。

```
sqrt(pow(x,2) + pow(y,2));
Math.sqrt(Math. pow(x,2) + pow(y,2));
```

静态导入 Math 类中的所有 static 成员, 语句为: import static java.lang.Math.*, 若其中静态导入 Math 类中的 sqrt 方法, 语句为: import static java.lang.Math.sqrt。静态导入语句看起来和普通的 import 语句非常相似。但是, 普通 import 语句从某个包中导入了一个或所有的类, 而静态 import 语句从某个类中导入一个类的一个或所有的静态方法以及静态变量, 使得导入类的所有静态变量和静态方法在当前类直接可见, 使用这些静态成员无需再给出它们的类名。

【例 3.9】 使用静态导入语句。

StaticImport.java

```
package org.import1;
public class Simple {
    public static final String COUNTRY = "China";
    public static int add(int a, int b) {
        return a + b;
    }
}

package org.import2;
import static org.import1.Simple.COUNTRY;           // 静态导入
import static org.import1.Simple.add;
public class StaticImport {
    public static void main(String[] args) {
        System.out.println(add(3, 9));
        System.out.println(COUNTRY);
    }
}
```

程序运行结果:

```
12
China
```

3.6 内部类

在一个类内部定义的类叫做内部类或内置类(inner class)。内部类可以将逻辑上相关的一组类组织起来，并由外部类来控制内部类的可见性。当建立一个内部类时，其对象就拥有了与外部类对象之间的一种关系，这是通过 **this** 引用形成的，使得内部类对象可以随意访问外部类中的所有成员。把包含内部类的类称为外部类，在图 3.6 中，类 **Outer** 是外部类，类 **Inner** 是类 **Outer** 的内部类，**Test** 类会访问类 **Outer** 及它的内部类，可把 **Test** 类称为客户类。外部类只能处于 **public** 和默认访问级别，而成员内部类可以处于 **public**、**protected**、**private** 和默认这四种访问级别。

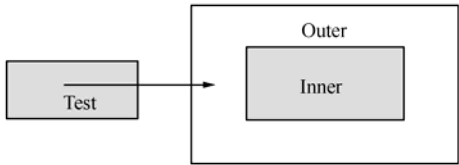


图 3.6 类的关系

3.6.1 实例内部类

实例内部类是成员内部类的一种，没有 **static** 修饰，实例内部类具有以下特点。

(1) 在创建实例内部类的实例时，外部类的实例必须已经存在。例如想创建内部类 **Inner** 类的实例，必须先创建外部类 **Outer** 类的实例，例如下面的语句：

```
Outer outer=new Outer(); // 创建外部类
Outer.Inner inner=outer.new Inner(); // 创建内部类
等价于：
```

```
Outer.Inner inner =new Outer().new Inner();
```

(2) 实例内部类的实例自动持有外部类的实例的引用，这个引用形式是：外部类名字.**this**。在内部类中，可以直接访问外部类的所有成员，包括成员变量和成员方法。

(3) 外部类实例与内部类实例之间是一对多的关系，一个内部类实例只会引用一个外部类实例，而一个外部类实例对应零个或多个内部类实例。在外部类中不能直接访问内部类的成员，必须通过内部类的实例去访问。

(4) 在实例内部类中不能定义静态成员，而只能定义实例成员，例如在以下内部类 **Middle** 中的三个静态成员，编译将无法通过。

```
class Outer {
    class Middle {
        static int a; // 编译错误
        int a; // 合法
        static void print(){} // 编译错误
        void print(){} // 合法
        static class Inner{} // 编译错误
        class Inner{} // 合法
    }
}
```

(5) 如果实例内部类与外部类包含同名的成员，例如，内部类变量 **Inner.a** 和外部类

Outer.a, 那么 this.a 表示内部类 Inner 的成员变量 a, Outer.this.a 表示外部类的变量 a。

【例 3.10】 实例内部类的特性。

Test.java

```
package org.innerclasses;
class Outer {
    private int index=100;
    private class Inner {
        private int index=50;
        void print() {
            int index=30;
            System.out.println(index);           ①
            System.out.println(this.index);       ②
            System.out.println(Outer.this.index); ③
        }
    }
    void print() {
        Inner i = new Inner();
        i.print();
    }
}
public class Test {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.print();
    }
}
```

程序运行结果：

```
30
50
100
```

说明：

- (1) 显示出 print()方法中的局部变量 index 的值；
- (2) 显示内部类 Inner 的私有变量 index 的值；
- (3) 显示外部类 Outer 的私有变量 index 的值。

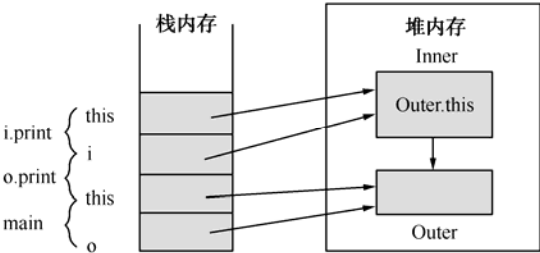


图 3.7 类的运行结构图

在 Test.java 程序的 main() 方法当中，首先定义了一个外部类变量 Outer，用 new 产生外部类的一个对象，由堆内存分配空间，如图 3.7 所示。它的引用保存在栈内存的 o 变量中。当调用外部类的 print() 方法，进入外部类的 print() 方法时，会有一个特殊变量 this 保存对象本身的一个引用。然后在 print() 方法当中用 new 又产生一个对象，进而在堆内存当中分配一个内部类对象，将它的引用保存到 i 变量中。用 i.print() 方法进入内部类 Inner 类的 print() 方法中，同样地，在 Inner 类的 print() 方法中，也有特殊的变量保存对象本身的一个引用，这里要注意到，在 Inner 对象中，还有一个 Outer.this 变量，保存 Outer 对象的一个引用。在内部类当中，访问外部类的所有成员就是通过 Outer.this 变量的。

3.6.2 匿名类

匿名类是一种特殊的类，这种类没有名字。匿名类具有以下特点：

(1) 匿名类是 final 类；

(2) 匿名类由于没有名字，因而无法定义构造方法，编译程序会自动生成匿名类的构造方法，在其中自动调用父类的构造方法；

(3) 在匿名类中可以定义实例变量和若干个实例初始化代码块和新的实例方法。Java 虚拟机首先调用父类的构造方法，然后按照实例变量的和实例初始化代码块定义的先后次序依次进行初始化。

(4) 匿名类除了可以继承类以外，还可以实现接口。

(5) 匿名类可以访问外部类的所有成员，如果匿名类位于一个方法中，还能访问所在方法的 final 类型的局部变量。

【例 3.11】 匿名类的特点。

Outer.java

```
package org.innerclasses.anonymous;
public class Outer {
    Outer(int v) {
        System.out.println("another constructor");
    }
    Outer() {
        System.out.println("default constructor");
    }
    void f() {
        System.out.println("from Outer");
    };
    public static void main(String args[]) {
        new Outer().f();                                // 显示 from Outer
        final int i = 1;
        Outer a = new Outer(i) {                        // 匿名类
            {
                System.out.println("initialize constructor");
            }
            void f() {
                System.out.println("from anonymous" + "    " + i);
            }
        }
    }
}
```

```
        };  
        a.f();  
    }  
}
```

// 显示 from anonymous

说明：以上“new Outer(){...}”定义了一个继承类 Outer 的匿名类，大括号内是类 Outer 的类体，“new Outer(){...}”返回匿名类的一个实例的一个引用。

程序运行结果：

```
default constructor  
from Outer  
another constructor  
initialize constructor  
from anonymous 1
```

第 4 章 Java面向对象编程（下）

继承和多态是 Java 语言不可缺少的组成部分。当创建一个类时，它总是继承某一个类。如果该类没有显式声明继承某一个类，那它则继承 Java 的标准根类 Object 类。而多态不但能够提高代码的可读性，还可以使得程序更易于扩展。

4.1 继承

继承是复用程序代码的有力手段。当多个类之间存在相同的属性和方法时，可以把这些相同的属性和方法抽取出来放到一个单独的类中，这个单独的类被定义为父类 Base 类。其他的类通过 extends 关键字来声明继承父类 Base 类，这些类自动拥有父类 Base 类中的能被继承的属性和方法，继承父类的类被定义为子类 Sub 类。

4.1.1 继承的定义

在 Java 语言中，用 extends 关键字来声明一个类继承了另一个类，其语法格式是：

```
<修饰符> class <子类名> extends <父类> {  
    ...  
}
```

例如，下面的代码片段定义了一个子类 son 类，继承了父类 farther 类。

```
class father {  
    ...  
}  
public class son extends father {  
    ...  
}
```

注意：Java 只支持单继承，例如，下面的 son 类试图继承两个类。

```
class grandFather {  
    ...  
}  
class farther {  
    ...  
}  
public class son extends farther, grandFather {           // 错误，不允许继承两个类  
    ...  
}
```

【例 4.1】 计算箱子的体积和重量。

DemoBoxWeight.java

```
class Box {  
    double width;    double height;    double depth;
```

```

Box(double w, double h, double d) {
    width = w;    height = h;    depth = d;
}
Box() {
    width = 0;    height = 0;    depth = 0;
}
double volume() {
    return width * height * depth;
}
}

class BoxWeight extends Box {                                // 继承 Box 类
    double weight;
    BoxWeight(double w, double h, double d, double m) {
        width = w; height = h; depth = d; weight = m;
    }
}

public class DemoBoxWeight {
    public static void main(String[] args) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 30, 58.5);
        BoxWeight mybox2 = new BoxWeight(3, 5, 8, 30.5);
        System.out.println("Volume of mybox1 is " + mybox1.volume());
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println("Volume of mybox2 is " + mybox2.volume());
        System.out.println("Weight of mybox2 is " + mybox2.weight);
    }
}

```

程序运行结果：

```

Volume of mybox1 is 6000.0
Weight of mybox1 is 58.5
Volume of mybox2 is 120.0
Weight of mybox2 is 30.5

```

4.1.2 初始化基类

当创建一个子类的对象时，该对象包含了一个基类对象。这个基类对象与用子类直接创建的对象是一样的。二者的区别在于，后者来自于外部，而基类对象被包装在子类对象内部。Java 虚拟机会确保在子类构造器中调用基类的构造器来初始化基类。

【例 4.2】 子类构造器 C 调用基类的构造器 A 来初始化基类。

C.java

```

class A{
    private int i;
    A(){
        System.out.println("invoke A constructor,i = " + i);
    }
}

class B extends A{

```

```

        private String s;
        B(){
            System.out.println("invoke B constructor,s = "+ s);
        }
    }
    public class C extends B{
        public C(){
            System.out.println("invoke C constructor");
        }
        public static void main(String[] args){
            new C();
        }
    }
}

```

程序运行结果：

```

invoke A constructor,i = 0
invoke B constructor,s = null
invoke C constructor

```

4.1.3 方法的重写

子类通过 `extends` 关键字声明继承了父类的属性和方法，但子类可能觉得从父类继承过来的方法不能满足自己的要求，怎么办呢？解决方法是子类可以重写（或覆盖）父类的方法。例如，在下面的代码片段中，子类重写了父类的 `run()` 方法。

```

class Animal{
    ...
    void run(){
        // 慢跑
    }
}
class Tiger extends Animal {
    ...
    void run(){
        // 快跑
    }
}

```

【例 4.3】 子类 `Employee` 重写父类的 `getInfo()` 方法。

TestOverWrite.java

```

package org.OverWrite;
class Person {
    private String name;    private int age;
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
}

```



```

    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public String getInfo() {
        return "Name: " + name + "\n" + "age: " + age;
    }
}
class Employee extends Person {
    private int salary;
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
    public String getInfo() {                // 重写父类的 getInfo()方法
        return "Name: " + getName() + "\nage: " + getAge() + "\nschool: " + salary;
    }
}
public class TestOverWrite {
    public static void main(String arg[]) {
        Employee employee = new Employee();
        employee.setName("Mary");
        employee.setAge(20);
        employee.setSalary(2000);
        System.out.println(employee.getInfo());
    }
}

```

程序运行结果:

```

Name: Mary
age: 20
school: 2000

```

在使用方法重写时，以下几点需要注意。

(1) 子类重写的方法必须与父类被重写的方法具有相同的方法名称、参数列表和相同或相容的返回值类型，否则不构成重写。例如，父类定义了方法：`int f(int i){...}`，若子类方法重写时写成：`byte f(int i){...}`，由于返回值类型是 Java 基本数据类型，必须要相同，因而编译程序会报错。但是，若父类定义了方法：`Object get(){...}`，子类方法重写时写成 `Point get(){...}`，虽然返回值类型不同，但由于 `Point` 是 `Object` 的子类，因而是允许的。即对于返回值类型是引用，则要求相容。这种方式的重写很有用。

(2) 子类重写的方法不能比父类中被重写的方法拥有更严格的访问权限。例如，在下面

的代码片段中，子类试图缩小父类方法的访问权限。

```
class Base{
    ...
    public void method() {
        ...
    }
}

public class Sub extends Base {
    ...
    private void method() {           // 编译错误，子类方法缩小父类方法的访问权限
        ...
    }
}
```

(3) 父类的静态方法不能被子类重写为非静态的方法。同样，父类中的实例方法也不能被子类重写为静态方法。例如下面的代码片段：

```
class Base{
    ...
    static void method() {
        ...
    }
}

public class Sub extends Base {
    ...
    void method() {                   // 编译错误
        ...
    }
}
```

- (4) 方法重写只针对实例方法，父类中的静态方法，子类只能隐藏、重载和继承。
- (5) 父类中能被子类继承的实例方法，才会在子类中被重写。

(6) 子类重写的方法不能比父类中被重写的方法声明抛出更多的异常。

方法重写和方法重载具有以下相同点。

- 都要求方法同名。
- 都可以用于抽象方法和非抽象方法之间。

方法重写和方法重载具有以下不同点。

- 方法重写要求参数签名必须一致，而方法重载要求参数签名必须不一致。
- 方法重写要求返回类型必须一致，而方法重载对此不做限制。
- 方法重写只能用于子类从父类继承的实例方法，方法重载用于同一个类的所有方法

(包括从父类中继承而来的方法)。

- 方法重写对方法的访问权限和抛出的异常有特殊的要求，而方法重载在这方面没有任何限制。
- 父类的一个方法只能被子类重写一次，而一个方法在所在的类中可以被重载多次。
- 构造方法能被重载，但不能被重写。

4.1.4 super关键字

若子类重写了父类中的方法或子类隐藏了父类中的数据成员，但又想访问父类的成员变量和方法，怎么办？解决的办法是使用 **super** 关键字。

【例 4.4】 子类 SubClass 使用 **super** 关键字，访问父类 SuperClass 的成员变量和构造方法。

TestSuperSub.java

```
class SuperClass {
    private int n;
    SuperClass() {
        System.out.println("SuperClass()");
    }
    SuperClass(int n) {
        System.out.println("SuperClass(" + n + ")");
        this.n = n;
    }
}
class SubClass extends SuperClass {
    private int n;
    SubClass(int n) {
        super(); // 访问父类的默认构造方法
        System.out.println("SubClass(" + n + ")");
        this.n = n;
    }
    SubClass() {
        super(100); // 访问父类的有参构造方法
        System.out.println("SubClass()");
    }
}
public class TestSuperSub {
    public static void main(String arg[]) {
        new TestSuperSub();
        SubClass sc1 = new SubClass();
        SubClass sc2 = new SubClass(200);
    }
}
```

程序运行结果：

```
SuperClass {100}
SubClass {}
SuperClass {}
SubClass {200}
```

4.2 对象的转型

对象既可以作为它自己本身的类型使用，也可以作为它的基类型使用。而这种把对某个

对象的引用视为对其基类型的引用的做法被称为向上转型，相反则称为向下转型。

例如，B 类是 A 类的子类或间接子类，当子类 B 创建一个对象，并把这个对象赋给类 A 的引用变量时，那么，称这个 A 类对象 a 是子类对象 b 的向上转型的对象。这个向上转型的对象还可以通过强制类型转换还原成它本来的类型，被称为对象的向下转型。

```
class A {}
class B extends A {}
A a;
B b1 = new B();
a = b1;                                // 向上转型
B b2 = (B) a;                          // 向下转型
```

向上转型的对象具有如下特点。

(1) 向上转型对象不能操作子类新增的成员属性和方法（失掉了这部分功能）。

(2) 向上转型对象可以操作子类继承或隐藏的成员变量，也可以使用子类继承的或重写的方法。

(3) 向上转型对象操作子类继承或重写的方法时，就是通知对应的子类对象去调用这些方法。因此，如果子类重写了父类的某个方法后，对象的向上转型对象调用这个方法时，一定是调用了这个重写的方法。

(4) 可以将向上转型对象再强制转换到它本来的类型，该对象又具备了其所有属性和方法。

【例 4.5】 测试对象转型的特点，使用 instanceof 判断一个实例对象是否属于某个类。

Cast.java

```
class Person {
    public String name;
    Person(String name) {
        this.name = name;
    }
}
class Student extends Person {
    public int studentId;
    Student(String str, int id) {
        super(str);                                // 调用父类的构造方法
        studentId = id;
    }
    void studying() {
        System.out.println("I am studying hard!");
    }
}
class Employee extends Person {
    public int salary;
    Employee(String str, int s) {
        super(str);                                // 调用父类的构造方法
        salary = s;
    }
}
```

```

    void working() {
        System.out.println("I am working hard!");
    }
}

public class Cast {
    public void testCast(Person p) {
        System.out.println("name is:" + p.name);
        if (p instanceof Student) {
            Student s = (Student) p;
            System.out.println("studentId is:" + s.studentId);
        } else if (p instanceof Employee) {
            Employee e = (Employee) p;
            System.out.println("studentId is:" + e.salary);
        }
    }

    public static void main(String[] args) {
        Cast cast = new Cast();
        Person p = new Person("Tom");
        Student s = new Student("John", 18);
        Employee e = new Employee("Lucy", 2000);
        System.out.println(p instanceof Person);           // true
        System.out.println(s instanceof Person);           // true
        System.out.println(e instanceof Person);           // true
        System.out.println(p instanceof Student);          // false
        p = new Employee("Mary", 3000);                    // 对象的向上转型
        System.out.println(p.name);
        // System.out.println(p1.salary);                   // 错误
        System.out.println(p instanceof Person);
        System.out.println(p instanceof Employee);
        Employee e1 = (Employee) p;                        // 对象的向下转型
        System.out.println(e1.salary);
        e1.working();
        p = new Student("Lily", 4000);                      // 对象的向上转型
        System.out.println(p.name);
        Student s1 = (Student) p;                          // 对象的向下转型
        System.out.println(s1.studentId);
        s1.studying();
        cast.testCast(p);
        cast.testCast(s);
        cast.testCast(e);
    }
}

```

`instanceof` 可以判断一个引用变量所指向的对象是否属于某个类，所以执行下面第一条语句返回 `true`，执行第二条语句发生了对象的向上转型，`Person` 类的引用变量指向 `Employee` 对象，如图 4.1 的箭头①所示。

```
System.out.println(p instanceof Employee);
p = new Employee("Mary", 3000);
```

但这时引用变量 **p** 所能访问的内容只限于 **Employee** 对象的父类 **Person** 对象，也就是箭头②所指向的区域。在执行下面的这条语句时，发生了对象的向下转型。

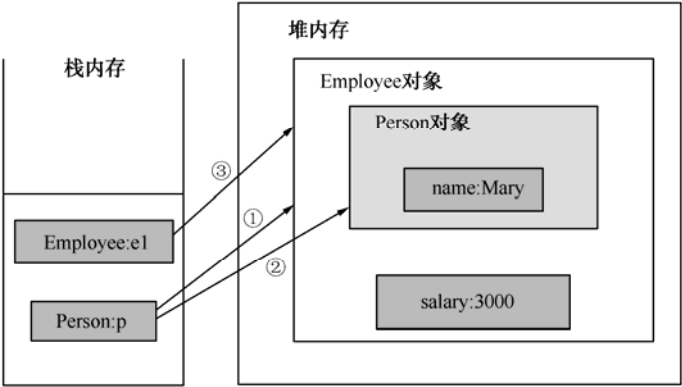


图 4.1 程序执行的内存布局

```
Employee e1 = (Employee) p;
```

Employee 类的引用变量 **e1** 同样指向 **Employee** 对象。这时引用变量 **e1** 能访问 **Employee** 对象的所有内容，也就是箭头③所指向的区域。

程序运行结果：

```
true
true
true
false
Mary
true
true
3000
I am working hard!
Lily
4000
I am studying hard!
name is:Lily
studentId is:4000
name is:John
studentId is:18
name is:Lucy
studentId is:2000
```

4.3 多态

运行 **Cast.java** 程序，可以发现 **testCase()**方法接受一个 **Person** 引用。那么在这种情况下，编译器怎样才能知道这个 **Person** 引用指向的是 **Student** 对象还是 **Employee** 对象？实际上，编译器无法得知，解决的方法就是动态绑定。它的含义就是在运行期间（而非编译期间）判断所引用对象的实际类型并根据对象的类型进行绑定，从而调用恰当的方法。只要满足类之间有继承关系、子类重写父类的方法、父类引用指向子类对象这 3 个条件，动态绑定就会自动发生，从而实现多态。多态不但能够改善代码的组织结构和可读性，还能够创建可扩展的程序，消除类型之间的耦合关系。多态方法调用允许一种类型表现出与其他相似类型之间的区

别，只要它们都继承同一基类。

【例 4.6】 Polymorphism 程序满足多态条件，根据对象的类型调用恰当的方法。

Polymorphism.java

```
package org.polymorphism;
class Student {
    String name;
    Student(String name) {
        this.name = name;
    }
    public void studying() {
        System.out.println("I am a student!");
    }
}
class UnderGraduate extends Student {
    private int credit;
    UnderGraduate(String n, int c) {
        super(n);
        credit = c;
    }
    public void studying() {
        System.out.println("I am a undergraduate!");
    }
}
class PostGraduate extends Student {
    private int paper;
    PostGraduate(String n, int c) {
        super(n);
        paper = c;
    }
    public void studying() {
        System.out.println("I am a postgraduate!");
    }
}
class Doctor extends Student {
    Doctor() {
        super("doctor");
    }
    public void studying() {
        System.out.println("I am a Doctor!");
    }
}
class Profession {
    private String name;
    private Student student;
    Profession(String name, Student s) {
        this.name = name;
```

```

        this.student = s;
    }
    public void tutor() {
        student.studying();
        System.out.println("my name is:" + student.name);
    }
}

public class Polymorphism {
    public static void main(String[] args) {
        UnderGraduate underGraduate = new UnderGraduate("Tom", 50);
        PostGraduate postGraduate = new PostGraduate("Jack", 5);
        Doctor doctor = new Doctor();
        Profession p1 = new Profession("DEK", underGraduate);
        Profession p2 = new Profession("DEK", postGraduate);
        Profession p3 = new Profession("DEK", doctor);
        p1.tutor();
        p2.tutor();
        p3.tutor();
    }
}

```

程序运行结果:

```

I am a undergraduate!
my name is:Tom
I am a postgraduate!
my name is:Jack
I am a Doctor!
my name is:doctor

```

说明 : UnderGraduate 类、PostGraduate 类、Doctor 类继承了父类 Student 类, 重写了父类的 studying()方法, 在创建 Profession 对象时, 分别把 UnderGraduate、PostGraduate、Doctor 对象传递给 student 引用, 从而满足多态条件。所以在调用 student.studying()方法时能够根据对象的类型调用恰当的方法。

4.4 抽象类

通过继承, 子类获得父类的变量和方法。考虑到更通用的情况, 例如在 Animal 类中定义了一个 run()方法, 子类也就自动拥有了该方法, 有一个袋鼠类 Kangaroo 类继承了 Animal 类, 也就拥有了 run()方法, 可袋鼠可能是跳跃式奔跑, Kangaroo 类不满意父类的 Animal 类的 run()方法, 当然 Kangaroo 类可以重写 run()方法。但是, 还有其他的类也继承了 Animal 类, 并且都要重新实现各自的 run()方法, 那么父类实现的 run()方法对于子类是没有意义的。这时考虑在父类中只声明 run()方法, 而不做任何的实现, 即没有方法体, 由子类实现 run()方法。

在 Java 中, 只声明而没有实现的方法称为抽象方法, 其语法规则如下。

```
abstract <返回值类型><抽象方法名>([<形式参数列表>]);
```


用 `abstract` 修饰的类称为抽象类，其语法规则如下。

```
[访问修饰符]abstract class <类名> {  
    ...  
}
```

下面的 `Graphix` 类定义为抽象类，方法 `calArea()` 定义为抽象方法。

```
abstract class Graphix {                                // 抽象类  
    abstract void calArea();                             // 抽象方法  
}  
class Rect extends Graphix {  
    void calArea(){                                     // 重写抽象方法  
        System.out.println("计算长方形面积");  
    }  
}
```

抽象类具有以下特性。

- (1) 含有抽象方法的类必须被声明为抽象类，抽象类必须被继承，抽象方法必须被实现。
- (2) 抽象类中不是所有的方法都是抽象方法，可以在抽象类中声明并实现方法。
- (3) 抽象类的子类必须实现父类的所有抽象方法后才能实例化，否则这个子类也成为一
个抽象类。
- (4) 抽象类不能实例化。

【例 4.7】 `Dog` 类继承 `Animal` 类并实现抽象方法 `run()`。

`TestAbstract.java`

```
abstract class Animal {  
    void sleep(){  
        System.out.println("animals sleep!");  
    }  
    abstract void run();                                // 抽象方法  
}  
class Dog extends Animal {  
    void run() {                                         // 实现抽象类  
        System.out.println("dogs run fast!");  
    }  
}  
public class TestAbstract {  
    public static void f(Animal a) {  
        a.sleep();  
        a.run();  
        Dog d;  
        d= (Dog)a;                                     // 向下转型  
        d.sleep();  
        d.run();  
    }  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        TestAbstract.f(d);  
    }  
}
```

```
}  
}
```

程序运行结果:

```
animals sleep!  
dogs run fast!  
animals sleep!  
dogs run fast!
```

4.5 接口

如果一个抽象类中的所有方法都是抽象的,而且这个抽象类中的数据成员都是 `final` 的常量,那么这个抽象类实际上就是一个接口,即一种特殊的抽象类。对于实现该接口的子类来说,接口不提供任何实现。接口是抽象方法和常量值定义的集合,而没有属性和方法的实现。

接口的定义格式如下。

```
[public]interface <接口名>[extends <一系列父接口>] {  
    <常量或抽象方法的集合>  
}
```

关键字 `interface` 用于定义接口,接口通常都定义为 `public` 类型。例如,定义一个接口:

```
public interface Runner {  
    int id = 1; // 等价于 public static final int id = 1;  
    public void start(); // 等价于 public abstract void start();  
    public void run();  
    public void stop();  
}
```

接口具有以下特性。

- (1) 接口中的常量默认为 `public static final`,并且也只能是 `public static final`。
- (2) 接口中只能定义抽象方法,而且这些方法默认为 `public abstract`,并且也只能是 `public abstract` 类型。
- (3) 接口可以继承其他的接口,并添加新的属性和抽象方法。
- (4) 在接口中声明方法时,不能使用 `native`、`static`、`final`、`synchronized`、`private`、`protected` 等修饰符。
- (5) `Java` 中不允许类的多继承,但允许接口的多继承。
- (6) 不允许创建接口的实例,但允许定义接口类型的引用变量,该变量引用实现了该接口的类的实例。
- (7) 一个类只能继承另外一个类,但能同时实现多个接口,并且重写的方法必须显式声明为 `public`。

【例 4.8】 测试接口的多种特性。

TestInterface.java

```
package org.interfaceImp;  
interface A{ // 接口  
    void a_f();  
}
```

```

interface B{
    void b_f();
}
interface C extends A,B{           // 接口的多继承
    void c_f();
}
class D {
    void d_f(){
        System.out.println("d_f()");
    }
}
class H implements A{
    public void a_f(){               // 必须显式声明为 public
        System.out.println("a_f()");
    }
    void h_f(){
        System.out.println("h_f()");
    }
}
class E extends D implements A,B{   // 多个接口
    public void a_f(){               // 须显式声明为 public
        System.out.println("a_f()");
    }
    public void b_f(){
        System.out.println("b_f()");
    }
}
public class TestInterface {
    public static void main(String[] args ){
        E e =new E();
        e.a_f();
        e.b_f();
        A a = new H();               // 接口的引用变量指向实现接口的类的实例
        a.a_f();
    }
}

```

程序运行结果：

```

d_f()
a_f()
b_f()
a_f()

```

4.6 final关键字

`final` 具有“不可改变”的含义，它可以修饰非抽象类、非抽象成员方法和变量。

- (1) 用 `final` 修饰的类不能被继承，没有子类。
- (2) 用 `final` 修饰的方法不能被子类的方法重写或隐藏。
- (3) 用 `final` 修饰的变量表示常量，只能被赋值一次。
- (4) 父类中用 `private` 修饰的方法不能被子类的方法重写，因此 `private` 类型的方法默认是 `final` 类型的。

【例 4.9】 测试 `final` 关键字。

TestFinal.java

```
final class T {                                     // final 类
    int i = 3;   int j = 6;
    void f() {
        System.out.println(i+j);
    }
}

public class TestFinal {
    private final int a = 12;                        // final 常量
    final void method() {                            // final 方法
        System.out.println("final method!");
    }
    public static void main(String[] args) {
        TestFinal test = new TestFinal();
        System.out.println(test.a);
        test.method();
        T n = new T();
        n.f();
    }
}
```

程序运行结果：

```
12
final method!
9
```

4.7 权限修饰符

Java 中的 `public`、`protected`、`private` 和默认（不加访问修饰符时）这几个访问修饰符置于类的每个成员（包括成员变量和成员方法）前，用来控制不同的访问权限。`public` 访问权限最大，`private` 访问权限最小。不过，即使是同一修饰词修饰成员变量和成员方法，在不同的情况下，其可见性范围也是不一样的。

1. `private` 访问权限

如果一个成员方法或成员变量使用了 `private` 访问控制符，那么这个成员只能在这个类的内部使用，其他类不能访问。

2. 默认访问权限

如果一个成员方法或成员变量没有使用任何访问控制符，就称这个成员是默认的或者包类型的。对于默认的访问控制成员，可以被这个包中的其他类访问。位于同一个包中的子类可以访问父类中的默认访问控制成员。但如果子类与父类位于不同的包中，子类则不能访问父类中的默认访问控制成员。

3. 受保护的访问权限

如果一个成员方法或成员变量前使用了 `protected` 访问控制符，那么这个成员既可以被同一个包中的其他类访问，也可以被位于不同包继承此父类的子类访问，但不能被不同包的类访问。

4. 公有的访问权限

如果一个成员方法或成员变量前使用了 `public` 访问控制符，那么这个成员可以被所有的类访问，不管访问类与被访问的类是否位于同一个包中，以及是否有继承关系。

4.7.1 类内部

在一个类的内部，其定义的 4 种不同的成员变量和成员方法，在整个类中都是可见的。

【例 4.10】 定义 4 种不同的成员变量和方法。

TestAccess1.java

```
package org.approach1;
public class TestAccess1 {
    private String var1="private variable";           // 私有成员变量
    String var2="default variable";                   // 缺省成员变量
    protected String var3="protected variable";       // 受保护的成员变量
    public String var4="public variable";              // 公有的成员变量
    private void f1(){                                // 私有成员方法
        System.out.println("private method");
    }
    void f2(){                                         // 缺省的成员方法
        System.out.println("default method");
    }
    protected void f3(){                              // 受保护的成员方法
        System.out.println("protected method");
    }
    public void f4(){                                  // 公有的成员方法
        System.out.println("public method");
    }
    public static void main(String[]args){
        TestAccess1 access1= new TestAccess1();
        System.out.println(access1.var1);
        System.out.println(access1.var2);
        System.out.println(access1.var3);
    }
}
```

```

        System.out.println(access1.var4);
        access1.f1();
        access1.f2();
        access1.f3();
        access1.f4();
    }
}

```

程序运行结果：

```

private variable
default variable
protected variable
public variable
private method
default method
protected method
public method

```

4.7.2 同一个包的类

除了 `private` 类型的所有成员变量和成员方法，一个类可以访问位于同一个包中的另一个类中的成员变量和成员方法。

【例 4.11】 访问位于同一个包中的其他类的成员变量和成员方法。

TestAccess2.java

```

package org.approach2;
import org.approach1.TestAccess1;
class Access3 {
    private String var1="private variable";
    String var2="default variable";
    protected String var3="protected variable";
    public String var4="public variable";
    private void f1(){
        System.out.println("private method");
    }
    void f2(){
        System.out.println("default method");
    }
    protected void f3(){
        System.out.println("protected method");
    }
    public void f4(){
        System.out.println("public method");
    }
}

public class TestAccess2 {
    public static void main(String[] args) {
        Access3 access3 = new Access3();
        //System.out.println(access3.var1);           // 不能访问另一个类的私有成员变量
        System.out.println(access3.var2);
    }
}

```

```

        System.out.println(access3.var3);
        System.out.println(access3.var4);
        //access3.f1();           // 不能访问另一个类的私有成员方法
    access3.f2();
        access3.f3();
        access3.f4();
    }
}

```

程序运行结果：

```

default variable
protected variable
public variable
default method
protected method
public method

```

如果上面程序的 TestAccess2 类继承 Access3 类，则其运行结果与上面的相同。

4.7.3 不同包的子类

如果类 B 继承位于不同包的类 A，那么类 B 只能访问类 A 中 protected 与 public 类型的成员变量和成员方法，不能访问类 A 中 private 与 default 类型的成员变量和成员方法。

【例 4.12】 访问不同包的子类的成员变量和成员方法。

TestAccess3.java

```

package org.approach2;
import org.approach1.TestAccess1;
public class TestAccess3 extends TestAccess1{
    public static void main(String[] args) {
        TestAccess3 access3 = new TestAccess3();
        //System.out.println(access3.var1);           // 编译错误
        //System.out.println(access3.var2);           // 编译错误
        System.out.println(access3.var3);
        System.out.println(access3.var4);
        //access3.f1();                               // 编译错误
        //access3.f2();                               // 编译错误
        access3.f3();
        access3.f4();
    }
}

```

程序运行结果：

```

protected variable
public variable
protected method
public method

```

4.7.4 通用性

如果类 B 与类 A 位于不同包，并且两者之间并没有继承关系，那么类 B 只能访问类 A 中 public 类型的成员变量和成员方法，其他类型的都不能访问。

【例 4.13】 访问不同包的类的成员变量和成员方法。

TestAccess4.java

```
package org.approach2;
import org.approach1.TestAccess1;
public class TestAccess4 {
    public static void main(String[] args) {
        TestAccess1 access1 = new TestAccess1();
        //System.out.println(access1.var1);
        //System.out.println(access1.var2);
        //System.out.println(access1.var3);
        System.out.println(access1.var4);
        //access1.f1();
        //access1.f2();
        //access1.f3();
        access1.f4();
    }
}
```

程序的运行结果：

```
public variable
public method
```

现在对以上情况进行归纳，如表 4.1 所示。

表 4.1 权限修饰符

	private	default	protected	public
同一个类	√	√	√	√
同一个包中的类		√	√	√
子类			√	√
任何包中的类				√

4.8 综合实例：航班管理

某航空公司对航班进行管理，管理控制台用如下几条命令进行管理。

命令 1：create 命令

格式：create <航班名> <多少排座位> <每排座位数>

功能：创建一个航班。

命令 2：reserve 命令

格式：reserve <旅客名 1> <旅客名 2> ...

功能：旅客预订座位，返回预订号。要求：同一批的旅客必须安排在同一排且座位相邻，

每一位旅客返回一个预订号。预订号自己定义，只要不重号即可。若安排不下（条件不能满足），则显示反馈信息。

命令 3: **Cancel** 命令

格式: **Cancel** <预订号 1> <预订号 2>

功能: 取消预订。

命令 4: **list** 命令

格式: **list**

功能: 显示座位预订情况。

命令 5: **exit** 命令

功能: 退出程序。

思路：对问题进行分析，可发现该问题固有的对象有旅客、航班、用户界面 3 个。对旅客和航班再设计接口，规定好它们对外界提供的界面，从而使 **Java** 程序具有“组件”的“可替换性”。

实现上述功能需要定义两个接口和 3 个类，两个接口分别是旅客接口和航班接口，3 个类分别是旅客接口的实现类、航班接口的实现类和用户界面操作类。

(1) 定义旅客接口 **PassengerInterface.java**

```
package org.aeroplane;

public interface PassengerInterface {
    public String getName();           // 获取姓名
    public int getBookingNumber();     // 获取预订号
    public int getRow();              // 获取座位排数
    public int getSeatPosition();     // 获取座位号
}
```

(2) 实现该接口的旅客类 **Passenger.java**

```
package org.aeroplane;

public class Passenger implements PassengerInterface {
    private String names;
    private int bookingNumber;
    private int rows;
    private int seatPosition;
    public Passenger(String names, int bookingNumber, int rows, int seatPosition) {
        this.names = names;
        this.bookingNumber = bookingNumber;
        this.rows = rows;
        this.seatPosition = seatPosition;
    }
    public String getName() {
        return names;
    }
    public int getBookingNumber() {
        return bookingNumber;
    }
    public int getRow() {
```

```

        return rows;
    }
    public int getSeatPosition() {
        return seatPosition;
    }
}

```

(3) 定义航班接口 FlightInterface.java

```

package org.aeroplane;

public interface FlightInterface{
    public String getName();
    public int getBookingNumber();
    public int getRow();
    public int getSeatPosition();
}

```

(4) 实现航班接口的航班类 Flight.java

在航班类中，要实现创建座位、完成客户座位的预订和取消座位的预订。航班类的源程序如下。

```

package org.aeroplane;

public class Flight implements FlightInterface {
    private String flightName;           // 航班名
    private int row;                      // 座位排数
    private int rowLength;               // 每排座位数
    private int[] fail = { -1 };         // 返回预订号
    private Passenger[] passengerList;   // 预订座位的旅客

    public Flight(String flightName, int rows, int rowLength)throws Exception {
        if (flightName == null || flightName.trim().length() == 0 || rows <= 0 || rowLength <= 0)
            throw new Exception("Error");
        else {
            this.flightName = flightName;
            this.row = rows;
            this.rowLength = rowLength;
            this.passengerList = new Passenger[row * rowLength]; // 创建航班座位
            for (int i = 0; i < row * rowLength; i++)
                passengerList[i] = null; // 所有座位没有被预订
        }
    }

    /******* 预订航班座位*****
    public int[] reserve(String names[]) {
        ... //代码见下页
    }

    /******* 取消航班预订座位*****
    public boolean cancel(int bookingNumber) {
        ... //代码见下页
    }

    public Passenger[] getPassengerList() {
        return passengerList;
    }
}

```

```

    }
}

```

其中：

● 创建航班座位

```
passengerList = new Passenger[row * rowLength];
```

● 预订航班座位

```

public int[] reserve(String names[]) {
    if (names.length > rowLength)
        return fail;
    int i = 0, j = 0, k = 0;
    // true—能安排, false—不能安排
    boolean flag = false;
    // 在同一排找相邻的且没有被预订的座位, 座位个数是 names.length
    labelA: for (i = 0; i <= row - 1; i++) {
        for (j = 0; j <= rowLength - names.length; j++) {
            // 在本行从 j 到 j+names.length - 1 找这样的空座位
            for (k = j; k <= j + names.length - 1; k++) {
                if (passengerList[i * rowLength + k] != null)
                    break;
            }
            if (k > j + names.length - 1) {
                flag = true;
                break labelA;
            }
        }
    }
    if (!flag)
        return fail;
    // 从第 i 行第 j 列开始分配座位
    int[] bn = new int[names.length];
    for (k = j; k <= j + names.length - 1; k++) {
        bn[k - j] = i * rowLength + k + 1;
        passengerList[i * rowLength + k] = new Passenger(names[k - j], i * rowLength + k + 1, i, k);
    }
    return bn;
}

```

● 取消预订航班座位

```

public boolean cancel(int bookingNumber) {
    boolean Status = false;
    for (int i = 0; i < row * rowLength; i++) {
        if (passengerList[i] != null && bookingNumber == passengerList[i].getBookingNumber()) {
            Status = true;
            passengerList[i] = null;
            break;
        }
    }
}

```

```
return Status;
```

```
}
```

(5) 用户航班命令输入和处理 Client.java

在 Client.java 中，能够对控制台输入的命令进行分析并且进行相应的处理，命令功能包括客户预订航班座位，取消航班预订和客户寻求帮助。源程序如下。

```
package org.aeroplane;
import java.util.*;
import java.io.*;
public class Client {
    private String flightName = null;           // 航班名
    private int row = 0;                        // 座位排数
    private int rowLength = 0;                  // 每排座位数
    private Flight flight = null;               // 本次航班对象
    private String cmdString = null;            // 命令串
    private BufferedReader br = new BufferedReader(new InputStreamReader(
        System.in));                             // 获取控制台命令
    public static void main(String[] args) {
        new Client ().commandShell();           // 命令 shell
    }
    private void commandShell() {
        System.out.println("\n\n=====");
        System.out.println(" Command Shell V0.01 ");
        System.out.println(" type 'exit' command to exit.");
        while (true) {
            readCommand();                       // 读命令
            processCommand();                     // 处理命令
        }
    }
    //*****从控制台读入命令*****
    private void readCommand() {
        ...                                     //代码见 75 页的“对控制台输入的指令进行分析”
    }
    //*****分析命令串*****
    // cmds 用于保存命令的各个分量，如命令：create sk213 10 5
    private void processCommand() {
        ...                                     //代码见 75 页的“对控制台输入的指令进行分析”
    }
    //*****分隔命令串*****/
    private String[] command(String cmdStr) {
        ...                                     //代码见 75 页的“对控制台输入的指令进行分析”
    }
    private int readInt(String valstr) {        // 把字符串类型转换为整型
        int val = 0;
        try {
            val = Integer.parseInt(valstr);
        } catch (Exception e) {
```

```

        val = Integer.MIN_VALUE;
    }
    return val;
}

private void createCommand(String[] cmds) { // 判断命令是否正确
    if (cmds.length != 4) {
        System.out.println("create command error!");
    }
    else {
        flightName = cmds[1];
        row = readInt(cmds[2]);
        rowLength = readInt(cmds[3]);
        if (row <= 0 || rowLength <= 0) {
            System.out.println("create command parameters error!");
            flightName = null;
            row = 0;
            rowLength = 0;
        } else {
            try {
                flight = new Flight(flightName, row, rowLength); // 创建航班座位
                System.out.println("create Flight OK!");
            } catch (Exception e) {
                System.out.println(e);
            }
            flight = null;
            flightName = null;
            row = 0;
            rowLength = 0;
        }
    }
}

//*****预订航班座位*****
private void reserveCommand(String[] cmds) { // 用户名放在 cmds[1],cmds[2],...
    ... //代码见 76 页的“客户预订航班座位”
}

//*****取消预订航班座位*****
private void cancelCommand(String[] cmds) {
    ... //代码见 77 页的“客户取消预订航班座位”
}

private void listCommand(String[] cmds) {
    if (cmds.length != 1) {
        System.out.println("\nlist command format error!");
        return;
    }
    Passenger[] passengerlist = flight.getPassengerList();
    int flag = 0;
    System.out.println("Name Booking Number Row Seat Position ");

```

```

        System.out.println("-----");
        if (passengerlist == null || passengerlist.length <= 0)
            System.out.println("Now no seat is occupied!");
        else {
            flag = 0;
            for (int b = 0; b < passengerlist.length; b++) {
                if (passengerlist[b] != null) {
                    flag = 1;
                    System.out.println(formatStr(passengerlist[b].getName())
                        + formatStr(" " + passengerlist[b].getBookingNumber())
                        + formatStr(" " + passengerlist[b].getRow()) + formatStr(" "
                        + passengerlist[b].getSeatPosition()));
                }
            }
            if (flag == 0)
                System.out.println("Now no seat is occupied!");
        }
    }

    private String formatStr(String s) {                                // 返回字符串
        for (int i = 0; i < 16 - s.trim().length(); i++)
            s += ' ';
        return s;
    }
}

```

其中：

- 对控制台输入的指令进行分析

```

// *****从控制台读入命令*****
private void readCommand() {
    // 若还没有创建航班，提示先创建航班
    if (flightName == null) {
        System.out.println("*****");
        System.out.println("Please Create The Flight Data First!");
        System.out.println("Use command: create flight_name rows rowLenght <CR>");
        System.out.println("*****\n\n");
    }
    System.out.print("\nCOMMAND>");                                // 命令提示符
    try {
        cmdString = br.readLine().trim();                            // 读取命令串
    } catch (IOException e) {
        System.out.println(" command error! ");
        cmdString = null;
    }
}

// *****分析命令串*****
// cmds 用于保存命令的各个分量，如命令：create sk213 10 5
private void processCommand() {

```

```

// "create" 放在 cmds[0], "sk213" 放在 cmds[1], "10" 放在 cmds[2], "5" 放在 cmds[3]
String[] cmds;
String cmd;
if (cmdString != null) {
    cmds = command(cmdString);                                // 分析命令, 分离出各个分量
    if (cmds != null) {
        cmd = cmds[0].toLowerCase();
        if (cmd.equals("create")) {                            // 处理 create 命令
            if (flightName == null)                             // 若航班还没有创建
                createCommand(cmds);
            else {                                              // 仅处理一个航班
                System.out.println("Create Error:can't handle more flights!");
            }
        } else if (cmd.equals("reserve")) {                    // 处理 reserve 命令
            if (flightName != null)                             // 当航班已创建过
                reserveCommand(cmds);
        } else if (cmd.equals("cancel")) {                     // 处理 cancel 命令
            if (flightName != null)                             // 当航班已创建过
                cancelCommand(cmds);
        } else if (cmd.equals("list")) {                       // 处理 list 命令
            if (flightName != null)                             // 当航班已创建过
                listCommand(cmds);
        } else if (cmd.equals("exit")) {                       // 处理 exit 命令
            System.out.println("Thanks. See you later!");
            System.exit(0);
        } else {
            System.out.println(" Bad command ! ");
            cmdString = null;
        }
    }
}

// *****分隔命令串*****
private String[] command(String cmdStr) {
    int cc = 0;                                                // 命令串中分量的个数
    String[] cmd;
    StringTokenizer st = new StringTokenizer(cmdStr);
    if ((cc = st.countTokens()) == 0)
        return null;
    cmd = new String[cc];
    for (int i = 0; i < cc; i++)
        cmd[i] = st.nextToken();
    return cmd;
}

```

● 客户预订航班座位

```

private void reserveCommand(String[] cmds) {                  // 用户名放在 cmds[1],cmds[2],...

```

```

    if (cmds.length <= 1) {
        System.out.println("reserve command error!");
        return;
    }
    String[] names = new String[cmds.length - 1];
    for (int i = 0; i < names.length; i++)
        names[i] = new String(cmds[i + 1]);
    int[] bn = flight.reserve(names);
    if (bn[0] != -1) {
        for (int i = 0; i < bn.length; i++)
            System.out.println(names[i] + "'s Booking Number is:" + bn[i]);
    } else
        System.out.println("No Such Sequential Seats Now!");
}

```

● 客户取消预订航班座位

```

private void cancelCommand(String[] cmds) {
    if (cmds.length != 2) {
        System.out.println("\ncancel command format error!");
        return;
    }
    int bookingNumber = readInt(cmds[1]);
    if (bookingNumber <= 0) {
        System.out.println("\ncancel command parameter error!");
        return;
    }
    boolean state = flight.cancel(bookingNumber);
    if (state)
        System.out.println(" Your seat has been cancelled! ");
    else
        System.out.println("The seat has not been reserved!");
}

```

● 运行程序，输入以下命令：

```
create sk213 10 5
```

程序运行结果：

```

COMMAND>create sk214 10 4
create Flight OK!

COMMAND>list
Name Booking Number Row Seat Position
-----
Now no seat is occupied!

COMMAND>reserve sk214 10 4
sk214's Booking Number is:1
10's Booking Number is:2
4's Booking Number is:3

COMMAND>

```


第 5 章 常用类与异常处理

Java 系统提供了大量的类和接口供程序开发人员使用，并且按照功能的不同，存放在不同的包中。这些包的集合就是应用程序接口（API），也称为“类库”。Java 包分核心包（Java core package）和扩展包（Java extension package）。核心包名以 java 开头，扩展包名以 javax 开头。Java 的类库非常庞大，被频繁使用的类有 Object、String、StringBuffer、日期类、基本数据类型的包装类等。

5.1 Object类

Java.lang.Object 类是所有 Java 类的最终祖先，如果一个类没有使用 extends 关键字显式声明继承某个类，那么这个类默认继承 Object 类。

```
public class Person {  
    ...  
}
```

等价于：

```
public class Person extends Object {  
    ...  
}
```

Object 类的主要成员方法有：equals()、hashCode()、toString()方法等。

5.1.1 equals()方法

equals()方法用于判断一个对象是否等于另外一个对象，实际上是比较两个引用是否指向同一个对象。如果两个引用指向同一个对象，equals()方法才返回 true。其方法的源代码为：

```
public boolean equals(Object obj){  
    return (this==obj);  
}
```

在实际编程中，更关心被比较的两个引用所指向的对象的状态（或属性）是否相同。许多 Java 类都重写了这个方法，如 String、Data、基本数据类型的包装类。

运算符“==”用来比较两个运算对象是否相等，这两个运算对象既可以是基本类型，也可以是引用类型。当两个运算对象都是引用类型时，那么这两个引用变量必须都引用同一个对象才返回 true。“==”用于 String 对象表示比较的是是否是同一个串对象，当用于比较两个字符串是否相等时，应使用 equals()方法。

【例 5.1】 综合运用“==”和 equals()方法。

TestEqual.java

```
public class TestEqual {  
    public static void main(String[] args) {  
        String s1 = "hello";    String s2 = "hello";
```

```

        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
        String s3 = new String("hello");
        String s4 = new String("hello");
        System.out.println(s3 == s4);
        System.out.println(s3.equals(s4));
        Person p1 = new Person("Tom", 18);
        Person p2 = new Person("Tom", 18);
        System.out.println(p1 == p2);
        System.out.println(p1.equals(p2));
        Person p3 = new Person("Lucy", 20);
        Person p4 = new Person("Jack", 20);
        System.out.println(p3 == p4);
        System.out.println(p3.equals(p4));
    }
}
class Person {
    String name;    int age;
    Person(String str, int i) {
        name = str;    age = i;
    }
    public boolean equals(Object obj) {           // 重写 Object 类的 equals()方法
        Person p = null;
        if (obj instanceof Person)
            p = (Person) obj;
        else
            return false;
        if (p.name == this.name && p.age == this.age)
            return true;
        else
            return false;
    }
}

```

说明：在程序中创建了两个 String 对象 s3 和 s4，由于 String 类重写了 equals()方法，所以 s3.equals(s4)比较的是两个对象的状态是否相同。同样，在 Person 类中重写了 equals()方法，p3.equals(p4) 比较的是两个对象的状态是否相同。

程序运行结果：

```

true
true
false
true
false
true
false
false

```

5.1.2 hashCode()方法

散列码（hash code）是由对象导出的一个整型值，主要是将对象作为 key，用于 Hash 表中，通常需要子类对该方法进行重写。常见的 String 类及基本数据类型的包装类如 Integer、Long 类等都已对 hashCode()和 equals()方法进行了重写，保证若 obj1.equals(obj2)，则 obj1.hashCode()==obj2.hashCode()。其含义是：若对象 obj1 和对象 obj2 按对象的状态（或属性或内容）是相等的，则它们的 hashCode 值应相同。即对象的 hashCode 值应由对象的状态决定。例如，两个不同的 String 对象只要包含相同的字符序列，其 hashCode 值就相同。

5.1.3 toString()方法

toString()方法返回对象的字符串表示，默认时，其格式为“类名@对象的十六进制哈希码”。其方法的源代码如下。

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

当 Java 系统处理对象时，每当需要将对象转成串时，都会自动调用该方法。当定义自己的对象时，应该重写该方法，以返回自己对象有明确含义串表示。许多 Java 类，如 Date、String、StringBuffer 和包装类都重写了 toString()方法，返回有实际意义的内容，例如下面的语句：

```
System.out.println(new Object().toString());           // 显示 java.lang.Object@10b30a7
System.out.println(new Integer(56).toString());        // 显示 56
System.out.println(new String ("hello").toString());   // 显示 hello
```

以上语句等价于：

```
System.out.println(new Object());                       // 显示 java.lang.Object@10b30a7
System.out.println(new Integer(56));                   // 显示 56
System.out.println(new String("hello"));               // 显示 hello
```

当 System.out.println()方法的参数是 Object 类型时，println()方法会自动调用 Object 对象的 toString()方法，然后显示 toString()方法返回的字符串。

【例 5.2】 使用 toString()方法显示字符串。

TestToString.java

```
class TestToString {
    private String s = "spring ";
    public TestToString(String str) {
        this.s = str + this.s;
    }
    public static void main(String[] args) {
        TestToString test = new TestToString("hibernate ");
        System.out.println(test);
        System.out.println(test.s);
    }
    public String toString() {                               // 重写 Object 类的 toString()方法
        this.s = "struts " + this.s;
        return s;
    }
}
```

程序运行结果:

```
struts hibernate spring
struts hibernate spring
```

5.2 字符串类

在Java中，一个字符串是一个Java对象，该对象封装了一个字符序列及有关该字符序列的其他信息，如长度等。java.lang包中的String、StringBuffer及StringBuilder类，分别用于处理常量字符串和长度与内容可变字符串，这3个类都被声明为final，因此都不能被继承。

5.2.1 String类

一个字符串常量是用双引号括住的一串字符，如"Hello"。一个字符串常量是一个String对象。Java.lang.String类代表只读的不可修改的字符序列，两个字符串对象使用“+”或“+=”运算符，会产生新的字符串对象。若Java程序中有多处出现字符串常量"Hello"，则Java编译程序只创建一个String对象，所有的字符串常量"Hello"将使用同一个String对象，例如下面的语句：

```
String s1="Hello";    String s2=" Hello";
String s3=new String("Hello");
String s4=new String("Hello");
```

则s1与s2是同一个对象，而s1、s3与s4是3个不同的对象，尽管它们所表示的字符序列相等。

String类的方法众多，有如下常用方法。

(1) int length(): 返回字符串的长度，例如下面的语句：

```
String s1 = "hello";
System.out.println(s1.length());           // 显示结果为 5
```

(2) char charAt(int index): 返回指定索引处的char值，其中index的取值范围是0~字符串长度-1，例如下面的语句：

```
String s1 = "hello world";
System.out.println(s1.charAt(6));           // 显示结果为 w
```

(3) int compareTo(String another): 按Unicode码值的大小逐字符比较两个字符串的大小。如果源串较小，则返回一个小于0的值，如果相等则返回0，否则返回一个大于0的值，例如下面的语句：

```
System.out.println("hello".compareTo("Hello"));           // 显示 -4
System.out.println("hello".compareTo("hello"));           // 显示 0
System.out.println("Hello".compareTo("hello"));           // 显示 1
```

(4) String concat(String str): 把字符串str附加在当前字符串的末尾，例如下面的语句：

```
String str = "hello";
String str2 = str.concat("world");
System.out.println("str");                     // 显示 hello
System.out.println("str2");                     // 显示 hello world
```

(5) equals(Object obj)和equalsIgnoreCase(String str): 判断两个字符串对象的内容是否相同。两个方法的区别在于，equals()方法区分字母的大小写，而equalsIgnoreCase()方法不区分

字母的大小写，例如下面的语句：

```
String str1="hello";    String str2="Hello";
System.out.println(str1.equals (str2));           // 显示 true
System.out.println(str1.equalsIgnoreCase (str2)); // 显示 false
```

(6) `int indexOf(int ch)`: 返回指定字符在此字符串中第一次出现处的索引。

`int indexOf(String str)`: 返回指定子字符串在此字符串中第一次出现处的索引。

`int lastIndexOf(int ch)`: 返回指定字符在此字符串中最后一次出现处的索引。

`int lastIndexOf(String str)`: 返回指定子字符串在此字符串中最右边出现处的索引。

例如下面的语句：

```
String s1 = "hello world";
System.out.println(s1.indexOf('l'));           // 显示 2
System.out.println(s1.indexOf("world"));       // 显示 6
System.out.println(s1.lastIndexOf('l'));       // 显示 9
System.out.println(s1.lastIndexOf("world"));   // 显示 6
```

(7) `String toUpperCase()`: 将此 `String` 中的所有字符都转换为大写。

`String toLowerCase()`: 将此 `String` 中的所有字符都转换为小写。

`String trim()`: 返回字符串的副本，忽略前导空白和尾部空白。

例如下面的语句：

```
String s1 = "Welcome to Java world";
String s2 = "  hello world  ";
System.out.println(s1.toUpperCase());           // 显示 WELCOME TO JAVA WORLD
System.out.println(s1.toLowerCase());          // 显示 welcome to java world
System.out.println(s2.trim());                 // 显示 hello world
```

(8) `String substring(int beginIndex)`: 返回一个新的字符串，该子字符串从指定索引处的字符开始，直到此字符串末尾。

`String substring(int beginIndex,int endIndex)`: 返回一个新字符串，该子字符串从指定的 `beginIndex` 处开始，直到索引 `endIndex - 1` 处的字符。

例如下面的语句：

```
String s1 = "Welcome to Java world";
System.out.println(s1.substring(11));          // 显示 Java world
System.out.println(s1.substring(11, 15));      // 显示 Java
```

(9) `static String valueOf()`: 把基本数据类型转换为 `String` 类型，例如下面的语句：

```
int i = 123;
String s1= String .valueOf(i);
System.out.println(s1);                        // 显示字符串 123
```

(10) `String[] split(String regex)`: 将一个字符串按照指定的分隔符分隔，返回分隔后的字符串数组。

【例 5.3】 按指定格式分隔字符串。

StringSplit.java

```
public class StringSplit {
    public static void print(String[] s) {
        for (int i = 0; i < s.length; i++)
            System.out.print(s[i] + " ");
    }
}
```

```

        System.out.println();
    }
    public static void main(String[] args) throws Exception {
        String[] result;
        String map = "value = hello";
        result = map.split("=");                // result={"value","hello"}
        print(result);
        String time = "08:30:36";
        result = time.split(":");                // result={"08","30","36"}
        print(result);
        String student = "Tom,20,university";
        result = student.split(",");            // result={"Tom","20","university"}
        print(result);
    }
}

```

程序运行结果:

```

value    hello
08 30 36
Tom 20 university

```

5.2.2 StringBuffer类

与 `String` 类表示的字符串不同, `StringBuffer` 类表示内容与长度随时动态可变的字符串缓存, 可直接对缓存进行插入、删除、修改、替换等操作, 操作结果影响串对象。在 Java 中通常不用字符数组来处理字符串, 其中一个原因是当对字符数组中的字符串进行处理时, 如不断插入新的子串, 会发生原先的字符数组空间不够的现象。怎么办? `StringBuffer` 类会自动管理这类空间不够的情况, 程序员不必担心。

`StringBuffer` 对象表示字符串缓存时, 有两个基本概念, 一个是容量, 另一个是字符串的长度。长度是 `StringBuffer` 对象所表示的字符串的长度, 容量是 `StringBuffer` 对象为存放字符串所拥有的空间大小。例如, `StringBuffer` 对象的容量是 100, 但其存放的字符串长度是 10。

`StringBuffer` 的构造方法如下。

- `public StringBuffer()`: 构造一个不带字符的字符串缓冲区, 其初始容量为 16 个字符。
- `public StringBuffer(int capacity)`: 构造一个不带字符, 但具有指定初始容量的字符串缓冲区。

- `public StringBuffer(String str)`: 构造一个字符串缓冲区, 并将其内容初始化为指定的字符串内容。该字符串的初始容量为 16 加上字符串参数的长度。

`StringBuffer` 的常用方法如下。

- `StringBuffer append()`: 向缓冲区内添加新的字符串。
- `StringBuffer insert(int offset,String str)`: 在字符串的 `offset` 位置插入字符串 `str`。
- `StringBuffer delete(int start,int end)`: 移除此序列的子字符串中的字符, 该子字符串从指定的 `start` 处开始, 一直到索引 `end - 1` 处的字符。
- `StringBuffer reverse()`: 将字符序列逆序。

String 类与 StringBuffer 类的区别如下。

(1) String 类代表字符串，Java 程序中的所有字符串字面值（如"abc"）都作为此类的实例实现，字符串是常量，它们的值在创建之后不能更改，因此，String 类的对象都是线程安全的。StringBuffer 代表线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，通过某些方法调用可以改变该字符序列的长度和内容，如 append()、insert()、delete()、replace() 方法等。

(2) String 类重写了从 Object 类所继承的 equals()方法，而 StringBuffer 类没有重写该方法，例如下面的语句：

```
String s1 = new String("hello");  String s2 = new String("hello");
System.out.println(s1.equals(s2));                                // 显示 true
StringBuffer sb3 = new StringBuffer("hello");
StringBuffer sb4 = new StringBuffer("hello");
System.out.println(sb3.equals(sb4));                                // 显示 false
```

(3) String 对象之间可以用操作符 “+” 进行连接，而 StringBuffer 对象之间不能通过操作符进行连接，但可使用 append()方法，例如下面的语句：

```
String s1 = "spring";  String s2 = "hibernate";
System.out.println(s1+s2);                                        // 显示 springhibernate
StringBuffer sb3 = new StringBuffer("spring");
System.out.println(sb3.append("hibernate"));                      // 显示 springhibernate
```

(4) StringBuffer 或 StringBuilder 是不能比较大小的，而 String 是可以比较大小的。因此可以方便地将 StringBuffer 转成 String 再比较大小。

(5) 若一个字符串有频繁的插入、删除、修改等操作，则适宜使用 StringBuffer（线程安全）或 StringBuilder（与 StringBuffer 相同，只是用于单线程下，性能更好），否则可使用 String 类进行大小比较、字符串结构分析等。若对一个 String 对象进行频繁的插入、删除、修改等操作，会产生大量的临时 String 对象，内存空间消耗较大。因此应按操作需求随时在 StringBuffer 与 String 之间互相转换。

(6) String 类 “+” 运算符实际上是由 StringBuilder 的 append()方法来实现的。

(7) 在程序设计时，经常需要 String 与其他类型之间进行类型转换，转换方法如表 5.1 所示。

表 5.1 String 与其他类型之间的类型转换

类 型	转到 String	从 String 转出
boolean	String.valueOf(boolean)	Boolean.parseBoolean(String)
byte	String.valueOf(int)	Byte.parseByte(String,int base)
char	String.valueOf(char)	str.charAt(o)
short	String.valueOf(int)	Short.parseShort(String,int base)
int	String.valueOf(int)	Integer.parseInt(String,int base)
long	String.valueOf(long)	Long.parseLong(String,int base)
float	String.valueOf(float)	Float.parseFloat(String)
double	String.valueOf(double)	Double.parseDoule(String)

【例 5.4】 测试 StringBuffer 类的常用方法 append()、insert()、delete()。

TestStringBuffer.java

```
public class TestStringBuffer {
    public static void main(String[] args){
        String s = "123";
        char[] a = {'a','b','c'};
        StringBuffer sb1 = new StringBuffer(s);
        sb1.append("struts").append("hibernate").append("spring");    //追加字符序列
        System.out.println(sb1);
        StringBuffer sb2 = new StringBuffer("中国");
        for(int i = 0;i< 10;i++){
            sb2.append(i);
        }
        System.out.println(sb2);
        sb2.delete(5, sb2.length());    //移除字符序列
        System.out.println(sb2);
        sb2.insert(3,a);    //插入字符序列
        System.out.println(sb2);
        System.out.println(sb2.reverse());    //逆序
    }
}
```

程序运行结果：

```
123strutshibernatespring
中国0123456789
中国012
中国0abc12
21cba0国中
```

5.3 包装类

Java 既提供了基本数据类型，也提供了相应的包装类。使用基本数据类型，能够满足大多数应用需求，但基本数据类型不具有对象的特性，不能满足特殊需求。Java 中的很多类的不少方法的参数类型是 Object 类型，即这些方法接收的参数都是对象，而又需要用这些方法处理基本数据类型的数据，这时包装类就很有用了。表 5.2 列出每一个基本数据类型对应的包装类。

表 5.2 基本数据类型与包装类

基本数据类型	对应的包装类
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

包装类具有以下特点。

(1) 所有的包装类都是 **final** 类型，因此不能创建它们的子类。

(2) 包装类是不可变类，一个包装类的对象创建后，它所包含的基本数据类型就不能被改变了。因此，所有包装类的对象都是线程安全的。

(3) 从 **JDK5.0** 版本开始，提供了自动装箱和自动拆箱机制。例如下面的语句：

```
Integer oi = 3;           // 自动装箱，等价于 Integer oi = Integer.valueOf(3);
int i  =oi                // 自动拆箱，等价于 int i = oi.intValue();
int v = oi + 1;           // 混合运算，等价于 int v = Integer.valueOf(oi.intValue()+1);
```

(4) 将字符串转换为基本值的 **parseType** 方法，例如，**Integer.parseInt(args[0])**可获得命令行上提供的 **int** 型数据。

(5) 可生成该对象基本值的 **typeValue** 方法 (**a.intValue()**)。

Character 类和 **Boolean** 类直接继承 **Object** 类，除此以外，其他包装类都是 **java.lang.Number** 的直接子类，因此都继承或者重写了 **Number** 类的方法，如图 5.1 所示。

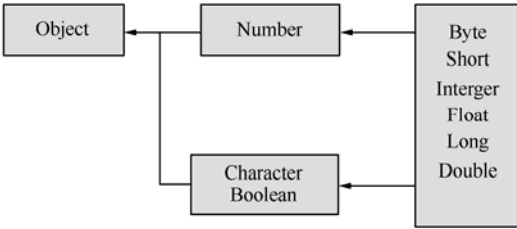


图 5.1 包装类的类层次结构

(6) 所有的包装类都可以用和它对应的基本数据类型作为参数，来构造它们的实例，例如下面的语句：

```
Boolean b = new Boolean(true);           // 或: Boolean b = true;
Byte bt = new Byte((byte)10);            // 或: Byte b = 10;
Character c = new Character('b');         // 或: Character c = 'b';
Integer i = new Integer(1);               // 或: Integer i = 1;
Float f = new Float(3.14f);               // 或: Float f = 3.14 f;
```

(7) 包装类都重写了 **Object** 类的 **toString()**方法，以字符串的形式返回包装类对象所表示的基本数据类型。

(8) 除 **Character** 类和 **Boolean** 类以外，包装类都有 **valueOf(String str)**静态工厂方法，可以根据 **String** 类型的参数来创建包装类对象。参数字符串 **str** 不能为 **null**，而且该字符串必须可以解析为相应的基本数据类型，例如下面的语句：

```
Double d = Double.valueOf("3.14");       // 合法
Integer i = Integer.valueOf("10");        // 合法
Integer i2 = Integer.valueOf("Tom");      // 抛出 NumberFormatException
```

(9) 除 **Character** 类以外，包装类都有 **parseType(String str)**静态方法，把字符串转换为相应的基本数据类型数据，且该字符串必须是可以解析为相应的基本数据类型的数据，例如下面的语句：

```
int i = Integer.parseInt("10");           // 合法
int a = Integer.parseInt("true");         // 抛出 NumberFormatException
```

【例 5.5】 测试包装类的属性和方法。

TestWrap.java

```
import java.util.Random;                                // 引入随机数类
public class TestWrap {
    public static void main(String[] args){
        Integer i = new Integer(5);
        Random r1 = new Random();                        // 创建 Random 对象，用于生成伪随机数
        // 返回下一个 double 类型的伪随机数，伪随机数的值大于或等于 0，并且小于 1.0
        Double d = r1.nextDouble();
        System.out.println(r1.nextDouble());
        int j = i.intValue()+d.intValue();
        float f = i.floatValue()+d.floatValue();
        System.out.println(j);
        System.out.println(f);
        double pi = Double.parseDouble("3.14");
        double r = Double.valueOf("5.0").doubleValue();
        double s = pi*r*r;
        System.out.println(s);
    }
}
```

程序运行结果：

```
10
10.68
78.5
```

5.4 Math类

Java.lang.Math 类提供了许多用于数学运算的静态方法，包括指数运算、对数运算、平方根运算和三角运算等。Math 类还有两个静态常量：E（自然对数）和 PI（圆周率）。Math 类是 final 类型的，因此不能派生子类。另外，Math 类的构造方法是 private 类型的，因此 Math 类不能够被实例化。Math 类的方法较多，现列出部分方法，如表 5.3 所示。

表 5.3 Math 类的方法

方 法	描 述
abs()	返回绝对值
floor(double d)	返回最大的 double 值，该值小于等于参数，并等于某个整数
pow(double a,double b)	返回第一个参数的第二个参数次幂的值
random()	返回带正号的 double 值，该值大于等于 0.0 且小于 1.0
round(float a)	返回最接近参数的 int
round(double a)	返回最接近参数的 long
min(a,b)	返回两者中较小的一个
max(a,b)	返回两者中较大的一个

【例 5.6】 输入三角形的三边长，求此三角形面积。

TestMath.java

```
public class TestMath {
    public static void main(String[] args) {
        int a = 0;  int b = 0;  int c = 0;
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
        c = Integer.parseInt(args[2]);
        double area =0;  double s = 0.0;
        s = 1.0/2*(a+b+c);
        area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        System.out.println("三角形的面积是"+area);
    }
}
```

右击 “TestMath.java”，选择 “Run As” → “Run Configurations”，如图 5.2 所示，选择 Main 标签页，在 “Project” 栏中选择 “MyProject_05”，在 “Main class” 栏中选择 “TestMath”，选择 “Arguments” 标签页，在 “Program arguments” 栏中输入 “6 8 10”，然后单击 “Run” 按钮，运行程序。



图 5.2 计算三角形面积

程序运行结果：

三角形的面积是24.0

5.5 日期类

Java 语言最先用 Date 类处理日期和时间，最简单的构造方法是 Date()，它以当前的日期和时间初始化一个 Date 对象。后来考虑到国际化问题，新增加两个类：一个是 java.util.Calendar 类，另一个是 java.text.DateFormat 类。它们都是抽象类，其实现类分别是 java.util.GregorianCalendar、java.text.SimpleDateFormat。

Date 类用毫秒数表示特定的日期和时间。在 Java 中有两个 Date 类，一个是 java.util.Date，

另一个是 java.sql.Date 类, java.util.Date 是 java.sql.Date 的父类。java.sql.Date 有一个 valueOf() 方法用于将 JDBC 日期转义形式的字符串转换成 Date 值。例如下面的语句:

```
java.util.Date date2 = date1.valueOf("2009-03-04");           // 返回一个日期类型
System.out.println(date2);                                     // 2009-03-04
```

其中 s 是以 “yyyy-mm-dd” 形式表示日期的 String 对象。

java.text.SimpleDateFormat 用于定义日期的格式, 该类主要用于将字符串解析成日期, 以及将日期格式化成字符串。

```
Date date = new Date();
DateFormat fullDateFormat = DateFormat.getDateInstance(DateFormat.FULL, DateFormat.FULL);
System.out.println(fullDateFormat.format(date)); // 2009 年 7 月 2 日 星期四 下午 03 时 53 分 22 秒 CST
```

现在已经能够解析一个字符串日期对象了, 但是如何才能设置和获取日期数据的特定部分, 比如日期、小时或者分钟? 又如何在日期的这些部分加上或者减去值呢? 使用 Calendar 类可以满足这些要求。java.util.Calendar 类是抽象类, 其实现类是 java.util.GregorianCalendar。Calendar 类的 set() 和 get() 方法可用来设置和读取日期的特定部分, 比如年、月、日等; add() 方法可用来在日期的特定部分加上一些数值。

【例 5.7】 打印 10 个既是星期一又是 11 号的日期。

CalendarDate.java

```
import java.util.GregorianCalendar;
import java.util.Date;
import java.text.DateFormat;

public class CalendarDate {
    public static void main(String[] args) {
        //*****设定日期和时间格式*****
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.FULL);
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(new Date());
        System.out.println("System Date: " + dateFormat.format(cal.getTime()));
        cal.set(GregorianCalendar.DAY_OF_WEEK, GregorianCalendar.MONDAY);
        System.out.println("After Setting Day of Week to Friday: "
            + dateFormat.format(cal.getTime()));

        int monday11Count = 0;
        while (monday11Count < 10) {
            cal.add(GregorianCalendar.DAY_OF_MONTH, 7); //增加 7 天得到下一个星期一
            //*****判断星期一的那天是否是 11 号*****
            if (cal.get(GregorianCalendar.DAY_OF_MONTH) == 11) {
                monday11Count++;
                System.out.println(dateFormat.format(cal.getTime()));
            }
        }
    }
}
```

程序运行结果:

```
System Date: 2009年6月30日 星期二
After Setting Day of Week to Friday: 2009年6月29日 星期一
2010年1月11日 星期一
2010年10月11日 星期一
2011年4月11日 星期一
2011年7月11日 星期一
2012年6月11日 星期一
2013年2月11日 星期一
2013年3月11日 星期一
2013年11月11日 星期一
2014年8月11日 星期一
2015年5月11日 星期一
```

5.6 正则表达式

正则表达式是一种强大而灵活的文本处理工具。使用正则表达式能够以编程的方式，构造复杂的文本模式，并对输入的字符串进行搜索。一旦找到了匹配这些模式的部分，就能够对它们进行处理。正则表达式主要用于验证和解析。验证是指使用正则表达式来检验输入字符串是否符合一个给定的模式。例如，可以校验一个电话号码是否遵循了格式“(000)0000-0000”(这里 0 表示一个数字)。解析是指分析输入的字符串，并分解它所包含的要素。例如，用模式 “(*):(*)” 可以用来解析出在 HTTP 头中普遍应用的 Key:Value 格式的头信息。

应用程序常常需要有文本处理功能，如单词的查找替换、电子邮件的验证或 XML 文档的集成。这些操作通常会涉及模式匹配的问题。java.util.regex 开发包可用于正则表达式处理模式匹配的问题。在 regex 包中只有两个类 (Pattern 和 Matcher 类)。Pattern 编译正则表达式，它会根据 String 类型的正则表达式生成一个 Pattern 对象。接下来，把想要检索的字符串传入 Pattern 对象的 matcher()方法。matcher()方法会生成一个 Matcher 对象，它的 replaceAll()方法能将所有匹配的部分替换成传入的参数。

正则表达式用来描述特定的字符串模式，例如正则表达式 a{3}表示由 3 个字符“a”构成的字符串，相当于普通字符串“aaa”，正则表达式 “\d”表示一位数字。在正则表达式中，有些字符具有特殊含义，参见表 5.4。

表 5.4 正则表达式中特殊字符

特殊字符	描述
[abc]	a、b 和 c 的任意一个字符
[^abc]	除了 a、b 和 c 之外的任意字符（否定）
[a-zA-z]	从 a~z 或从 A~Z 的任意字符（范围）
[abc[hij]]	任意 a、b、c、h、i 和 j 字符（并集）
[a-z&&[hij]]	任意 h、i 或 j（交集）
\s	空白符（空格、tab、换行和回车）
\S	非空白符 ([^\s])
\d	数字[0-9]
\D	非数字[0-9]
\w	单词字符[a-zA-z]

特 殊 字 符	描 述
.	任何一个字符
\\	反斜线字符\
\uhhhh	十六进制表示的 unicode 值为 hhhh 的字符

下面对表 5.4 所列举的每一个特殊字符列举一个示例，如下所示。

```
System.out.println("b".matches("[abc]"));           // 打印 true
System.out.println("b".matches("[^abc]"));           // 打印 false
System.out.println("A".matches("[a-zA-Z]"));         // 打印 true
System.out.println("A".matches("[a-z][A-Z]"));       // 打印 true
System.out.println("A".matches("[a-zA-Z]"));       // 打印 true
System.out.println("R".matches("[A-Z&&[RFG]]"));     // 打印 true
System.out.println("\nt".matches("\s{2}"));         // 打印 true
System.out.println(" ".matches("\S"));              // 打印 false
System.out.println("a_8".matches("\w{3}"));         // 打印 true
System.out.println("3".matches("\d"));              // 打印 true
System.out.println("&".matches("\D"));              // 打印 true
```

表 5.5 同时列出了一些在正则表达式中量词所表示字符的出现次数。

表 5.5 量词

量 词	描 述
X*	0 个或多个 X（最大匹配）
X+	一个或多个 X（最大匹配）
X?	0 个或一个 X（最大匹配）
X{n}	恰好 <i>n</i> 个 X
X{n,}	至少 <i>n</i> 个 X
X{n,m}	至少 <i>n</i> 个 X，不多于 <i>m</i> 个 X
X*?	0 个或多个 X（最小匹配）
X+?	1 个或多个 X（最小匹配）
X??	0 个或多个 X（最小匹配）
XY	X 后跟 Y
X Y	X 或 Y
(X)	定义捕获组
\n	与第 <i>n</i> 个捕获组相匹配的字串

下面的位置匹配，用于匹配串中的位置。

- (1) (?=X)：与这样的位置相匹配，其右部能匹配 X。
- (2) (?!X)：与这样的位置相匹配，其右部不能匹配 X。
- (3) (?<=X)：与这样的位置相匹配，其左部能匹配 X。
- (4) (?<!X)：与这样的位置相匹配，其左部不能匹配 X。

运用以上正则式规则，可设计常用的正则表达式。例如：

- (1) 数字串：\d+，在 Java 中表示为“\\d+”。

- (2) 英文单词: [a-z A-Z]+, 在 Java 中表示为“[a-z A-Z]+”。
- (3) 一个汉字: [\u4e00- \u9fa5]在 Java 中表示为“[\u4e00-\u9fa5]”。
- (4) 汉字串: [\u4e00- \u9fa5]+在 Java 中表示为“[\u4e00-\u9fa5]+”。
- (5) IP 地址: (\d{1,3}\.){3}\d{1,3}在 Java 中表示为“(\d{1,3}\.){3}\d{1,3}”。
- (6) 重复字符压缩: 将字符串中连续重复的字符压缩成一个。例如:

“aaabbbcccd11122++***...33”压缩成“abcd12+*.3”。相关代码是: 设 String s =""; 则 String rs = s.replaceAll("(.)\\1*", "\$1");

说明: (.)表示一号组, 匹配任何一个字符, \\1 表示与一号组相同的字符, \$1 表示在 replaceAll()方法中, 与一号组相匹配的那个子串。

- (7) 特定子串提取: 例如,

String s =“%...%CXLL= add1,31,123.12%CXLL= add2,32,124%CXLL=,33,125.12%LL= -121.11”;
从中提取出: %CXLL= add1,31,123,123.12, %CXLL= add2,32,124.12, %CXLL= ,33,125.12。
提取的正则表达式设计成: %CXLL=.*?(?=%)。Java 中表示为“%CXLL=.*?(?=%)”。

说明: .*?表示以最小匹配方式去匹配任何串 (若没有? 表示最大匹配, 则内部将包含有 %)。(?=%)表示必须与这样的位置相匹配, 其右部有一个字符%。

- (8) 串格式: 不能以 bug 打头可以由字母、数字组成的 8~12 个数的字符串。

正则式: (?!bug)[a-zA-Z0-9]{8-12}

下面对表 5.5 所列举的每一个特殊字符列举一个示例, 如下所示。

```
System.out.println("aaaa".matches("a*"));           // 打印 true
System.out.println("aaaa".matches("a+"));           // 打印 true
System.out.println("".matches("a*"));               // 打印 true
System.out.println("aaaa".matches("a?"));           // 打印 false
System.out.println("".matches("a?"));               // 打印 true
System.out.println("aaaa".matches("a{4}"));         // 打印 true
System.out.println("abcabcabc".matches("(abc){2}")); // 打印 false
System.out.println("abcabcabc".matches("(abc){2,}")); // 打印 true
System.out.println("aa".matches("a?")+ "12");       // 打印 false
System.out.println("4563456257".matches("\\d{3,10}")); // 打印 true
```

一般来说, 比起功能有限的 String 类, Java 提供了 java.util.regex 包来构造功能强大的正则表达式对象。只需导入 java.util.regex 包, 然后用 static 型 Pattern.compile()方法来编译正则表达式即可。它会根据 String 类型的正则表达式生成一个 Pattern 对象。接下来, 把想要检索的字符串传入 Pattern 对象的 matcher()方法。matcher()方法会生成一个 Matcher 对象, 它的 replaceAll()方法能将所有匹配的部分替换成传入的参数。

Pattern 类的实例代表了一个以字符串形式指定的正则表达式。用字符串形式指定的正则表达式, 必须先编译成 Pattern 类的实例。生成的模式用于创建 Matcher 对象, Matcher 对象可以根据正则表达式与任意字符序列进行匹配。Pattern 类的一些方法可以完成简单的模式匹配, 有:

- static Pattern compile(String regex, int flags)

编译模式, 参数 regex 表示输入的正则表达式, flag 表示模式类型, 如 DOTALL CASE_INSENSITIVE、MULTILINE、UNICODE_CASE。

- Matcher matcher(CharSequence input)

创建与给定输入模式匹配的匹配器, input 表示待匹配的字符序列。

- static boolean matches(String regex,CharSequence input)

编译给定正则表达式并尝试将给定输入与其匹配。

Matcher 类的实例用于根据给定的模式，对字符串进行匹配。使用 CharSequence 接口把输入的字符串提供给匹配器，以便支持来自不同输入源的字符串的匹配。当创建了匹配器之后，就可以用它来执行 3 类不同的匹配操作。

- matches()方法试图根据此模式，对整个输入序列进行匹配。
- lookingAt()方法试图根据此模式，从开始处对输入序列进行匹配。
- find()方法将扫描输入序列，寻找下一个与模式匹配的地方。

【例 5.8】 使用正则表达式匹配输入的字符串。

TestRegex.java

```
package regex;
import java.util.regex.*;
public class TestRegex {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.exit(0);                // 退出程序
        }
        System.out.println("Input: \"" + args[0] + "\"");
        for (String arg : args) {
            System.out.println("Regular expression: \"" + arg + "\"");
            Pattern p = Pattern.compile(arg);
            Matcher m = p.matcher(args[0]);
            while (m.find()) {
                System.out.println("Match \"" + m.group() + "\" at positions "
                    + m.start() + "-" + (m.end() - 1));
            }
        }
    }
}
```

右击 “TestRegex.java”，选择 “Run As” → “Run Configurations”，如图 5.3 所示，选择 Main 标签页，在 “Project” 栏中选择 “MyProject_05”，在 “Main class” 栏中选择 “TestRegex”，选择 “Arguments” 标签页，在 “Program arguments” 栏中输入 “abcbabcbdefabc "abc+" "(abc)+" "(abc){2,}”，然后单击 “Run” 按钮，运行程序。

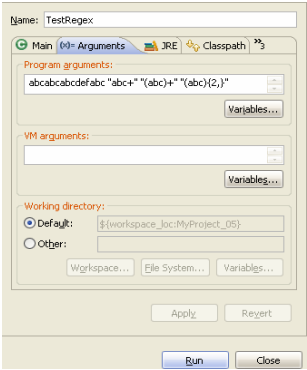


图 5.3 使用正则表达式匹配字符串

程序运行结果:

```
Input: "abcabcabcdefabc"
Regular expression: "abcabcabcdefabc"
Match "abcabcabcdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcabc" at positions 0-8
Match "abc" at positions 12-14
Regular expression: "(abc){2,}"
Match "abcabcabc" at positions 0-8
```

说明:第一个控制台参数用来搜索匹配的输入字符串,后面的多个参数都是正则表达式,它们将被用来在输入的字符串中查找匹配。

【例 5.9】 使用正则表达式设计一个 Java 程序,所要完成的功能是:对一个由数字和非数字组成的字符串,将其中连续的数字字符转换成一个整数。若连续的数字字符个数超过 4 个,则以 4 个数字字符为一组进行转换。转换生成的整数依次存放在整数数组中。例如,对字符串“c789yz45!786*+56abc123456789”,分析后的整数数组内容为:789、45、786、56、1234、5678、9。

GetNumber.java

```
package regex;
import java.util.regex.*;
public class GetNumber {
    public static void main(String[] args) {
        int[] arr = new int[10];           // 创建 1 个整型数组
        Pattern p = Pattern.compile("(\\d{1,4})"); // 编译正则表达式, 要求 1~4 个数字
        String s = "c789yz45!786*+56abc123456789";
        Matcher m = p.matcher(s);         // 对字符串进行匹配
        int i = 0;
        while(m.find()) {                 // 寻找与指定模式匹配的下一个子序列
            int j = 0;
            j = Integer.parseInt(m.group()); // 将字符串类型转换为整型
            arr[i] = j;
            i++;
        }
        for(int c = 0; c < i; c++){
            System.out.println(arr[c]);    // 打印数组的内容
        }
    }
}
```

程序运行结果：

```
789
45
786
56
1234
5678
9
```

5.6.1 替换操作

正则表达式擅长于替换文本，提供了多种方法用于实现这种功能。`replaceFirst(String replacement)`以参数字符串替换掉第一个匹配成功的部分。`replaceAll(String replacement)`以参数字符串 `replacement` 替换掉所有匹配成功的部分。`appendReplacement(StringBuffer buf,String replacement)`执行渐进式的替换，允许调用其他方法来生成或处理 `replacement`，能够以编程的方式将目标分隔成组，从而具有强大的替换功能。`appendTail(StringBuffer buf)`，在执行了一次或多次 `appendReplacement()`之后，调用此方法可以将输入字符串余下的部分复制到 `buf` 中。

【例 5.10】 使用正则表达式的替换功能将字符串的奇数序列的“java”替换为小写形式，将偶数序列的“java”替换成大写形式。

TestReplace.java

```
package regex;
import java.util.regex.*;
public class TestReplace {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("java", Pattern.CASE_INSENSITIVE); // 不区分大小写
        Matcher m = p.matcher("Java java JAvA JAVA I love JAVA you dislike Java ");
        StringBuffer buf = new StringBuffer();
        int i=0;
        while(m.find()) {
            i++;
            if(i%2 == 0) {
                m.appendReplacement(buf, "JAVA");           // 偶数序列的转换成大写
            } else {
                m.appendReplacement(buf, "java");           // 奇数序列的转换成小写
            }
        }
        m.appendTail(buf);
        System.out.println(buf);
    }
}
```

程序运行结果：

```
java JAVA java JAVA I love java you dislike JAVA
```

5.6.2 Scanner定界符

`Scanner` 的构造器可以接收任何类型的输入对象。有了 `Scanner`，所有的输入、分词及翻

译的操作都隐藏在不同类型的 `next()` 方法中。所有的 `next()` 方法，只有在找到一个完整的分词之后才会返回。`Scanner` 还有相应的 `hasNext()` 方法，用以判断下一个输入分词是否为所需的类型。在默认的情况下，`Scanner` 根据空白字符对输入进行分词。也可以用正则表达式指定自己所需的定界符。

【例 5.11】 使用 “.” 字符作为 IP 地址的定界符。

TestScanner.java

```
package regex;
import java.util.*;
public class TestScanner {
    public static void main(String[] args){
        Scanner scanner = new Scanner("192.168.1.99");
        scanner.useDelimiter("\\s*\\.\\s*");           // 使用"."作为定界符
        while(scanner.hasNextInt()){
            System.out.println(scanner.nextInt());
        }
    }
}
```

程序运行结果:

```
192
168
1
99
```

5.7 异常处理

所谓异常，就是以对象的方式表示的一个或一类错误。该异常对象不仅封装了错误信息，还包含了错误发生时的“上下文”信息。传统的错误处理方式是，每当程序调用一个方法进行某种操作，程序应立即检查该方法的返回值，判断是否有错误发生。若有错误发生，通常要访问一个全局变量如 `err` 来获取具体的错误代码，以得到进一步的错误信息。在 `Java` 中，这种传统的处理方式仍然有效，但 `Java` 从 `C++` 中继承了以面向对象的方式处理错误的机制，并做了进一步的改进。

在 `Java` 中，任何异常对象都是 `java.lang.Throwable` 类或其子类的对象，`Throwable` 类是 `Java` 异常类体系中的根类，如图 5.4 所示。它有两个子类：一个是 `Error` 类，另一个是 `Exception` 类。`Error` 类及其子类的对象，代表了程序运行时 `Java` 系统内部的错误，这样的错误程序员一般不用关心。因为一旦真的发生了这类错误，程序员除了告诉用户发生了错误并正常关闭程序的运行之外，也许没有其他更好的办法。`Exception` 类及其子类的对象是程序员应该认真关心并尽可能加以处理的。`Exception` 类有许多子类，其中如 `RuntimeException` 类表示 `Java` 程序运行时产生的错误，如数组下标越界，对象类型强制转换错误，空指针访问错误等。尽管这类运行错误程序员可以不必捕获，但它与 `Error` 类的错误性质是不同的。`RuntimeException` 类及其子类的对象，代表了程序员在设计程序时犯了错误，即程序内部有错误。程序员有责任改正程序，以求消除这类异常。`Exception` 类有相当多的子类，其中最常见的一个子类是 `IOException` 类。该类及其子类对象表示一个 I/O 错误，程序员应该在程序中进行处理。

所有异常分为两类：必须要检查的异常（checked exceptions）和非检查的异常（unchecked exceptions）。`Error` 和 `RuntimeException` 类及其子类都属于 `unchecked exceptions`，表示编译器

在编译程序时对该类异常不做检查。Exception 类及其子类（RuntimeException 除外）属于 checked exceptions，编译器在编译程序时，要检查程序是否对该类异常做了处理（如 try 捕获或 throws 声明抛出异常）；若没有处理，则编译器会报错，要求对此必须进行处理。

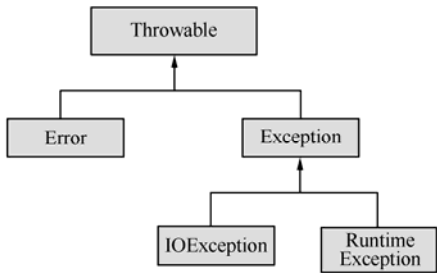


图 5.4 Java 异常类层次结构

5.7.1 异常的捕获与处理

Java 中使用 try-catch-finally 语句来捕获并处理异常，try-catch-finally 语句的语法格式如下。

```
try {
    // 可能会产生异常的程序代码
}catch(Exception_1 e1){
    // 处理异常 Exception_1 的代码
}catch(Exception_2 e2){
    // 处理异常 Exception_2 的代码
}
...
catch(Exception_n en){
    // 处理异常 Exception_n 的代码
}finally{
    // 通常是释放资源的程序代码
}
```

整个语句由 try 语句块、catch 语句块和 finally 语句块 3 部分组成。catch 语句块和 finally 语句块都是可以缺省的，但 Java 规范不允许两者同时缺省。

（1）try 语句块

将可能产生异常的程序代码放在此处，该段代码是程序正常情况下应该要完成的功能。在执行过程中，该段代码可能会产生并抛出一种或几种类型的异常对象，它后面的 catch 语句块要分别对这些异常做相应的处理。try 语句块后通常跟 0 个或多个 catch 语句块，还可以跟至多一个的 finally 语句块。

（2）catch 语句块

每个 catch 语句块声明其能处理的一种特定类型的异常并提供处理的方法。当异常发生时，程序会中止当前的流程，根据获取异常的类型去执行相应的 catch 语句块。在 catch 中声明的异常对象（catch(Exception e)）封装了异常事件发生的信息，在 catch 语句块中可以使用这个对象的一些方法来获取这些信息，例如下面的方法。

- getMessage(): 用来得到有关异常事件的信息。

● `printStackTrace()`：用来跟踪异常事件发生时执行堆栈的内容。

(3) `finally` 语句块

无论 `try` 所指定的程序块是否抛出异常，`finally` 所指定的代码都要被执行。`finally` 语句为异常处理提供一个统一的接口，使得在控制流程转到程序的其他部分以前，能够对程序的状态作统一的管理。通常在 `finally` 语句中进行资源的释放工作，如关闭打开的文件，删除临时文件等。

`try-catch-finally` 语句的语义：

首先执行 `try` 语句块中的代码，若一切正常，则跳过 `catch` 语句块，执行 `finally` 语句块中的代码（若该部分没有缺省）；执行完后，整个 `try-catch-finally` 语句才算执行完成。若执行 `try` 语句块时产生异常，则立即跳转到 `catch` 语句块。Java 虚拟机会把实际抛出的异常对象依次和各个 `catch` 语句块声明的异常类型匹配，如果异常对象为某个异常类型或其子类的对象，就执行这个 `catch` 语句块，而不会再执行其他的 `catch` 语句块。之后，跳转到 `finally` 语句块，执行完 `finally` 语句块，该语句才算执行完成。

【例 5.12】 算术运算异常的捕获和处理。

UseException.java

```
import java.io.*;

public class UseException {
    public static void main(String[] args) {
        try {
            int a, b;
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            System.out.println(a + " / " + b + " = " + (a/b));
        } catch (ArithmeticException e) {
            System.out.println("捕获了一个异常，除数不能为 0!");
        }
    }
}
```

右击 “UseException.java”，选择 “Run As” → “Run Configurations”，如图 5.5 所示，选择 Main 标签页，在 “Project” 栏中选择 “MyProject_05”，在 “Main class” 栏中选择 “UseException”，选择 “Arguments” 标签页，在 “Program arguments” 栏中输入 “20 5”，然后单击 “Run” 按钮，运行程序。



程序运行结果：

```
20 / 5 = 4
```

按同样步骤，当输入整数 0 时，运行程序。

程序运行结果：

```
捕获了一个异常，除数不能为0！
```

5.7.2 声明抛出异常子句

声明抛出异常是一个子句，只能加在方法头部的后边，语法格式如下。

`throws <异常列表>`

例如，`public int read() throws IOException { ... }`

若一个方法声明抛出异常，则表示该方法可能会抛出所声明的那些异常，从而要求该方法的调用者，在程序中对这些异常加以注意和处理，如用 `try-catch-finally` 语句来处理。若一个方法没有声明抛出异常，则该方法仍可能会抛出异常，只不过这些异常不要求调用者加以注意。即使一个方法声明抛出异常，该方法仍可能会抛出不在异常列表之中列出的异常。通常异常列表中的异常，都要求调用者在程序中明确加以处理。方式可以用 `try-catch-finally` 语句捕获处理，或者用 `try-catch-finally` 捕获处理后再次抛出异常，从而形成异常处理链。

【例 5.13】 从键盘读入字符，再输出到控制台。

ThrowsExceptionTest.java

```
import java.io.*;

public class ThrowsExceptionTest {
    public static void main(String[] args) throws IOException {
        int c;
        while ((c = System.in.read()) != -1)           // 从键盘读入数据
            System.out.println(c);
    }
}
```

运行程序，输入任意字符串“abcde”，按下回车键，则在控制台上输出其机内码。

程序运行结果：

```
abcde
97
98
99
100
101
13
10
```

说明：最后的数值 13 和 10 是回车和换行的值。因为 `System.in.read()` 方法会抛出 `IOException` 异常，而程序中没有用 `try-catch-finally` 语句进行捕获处理，故必须在 `main()` 方法的头部加上 `throws IOException`，明确表示对该异常，程序不想处理，交由调用者处理。当然 `main()` 方法的调用者是 JVM，一旦有异常，程序即运行结束。

注意：若一个异常是属于 checked exception，在方法内部没有用 `try` 语句捕获，则必须要

用 `throws` 子句在方法头中加以声明。如在例 5.13 中，`IOException` 异常是一个 `checked exception`，方法代码中又没有进行捕获，因此必须要声明抛出。`unchecked exception` 不必要如此做。

5.7.3 抛出异常语句

声明抛出异常是一个说明性的子句，只是表明一个方法可能会抛出异常，而真正抛出异常的动作是由抛出异常语句来完成的，语法格式如下。

```
throw <异常对象>;
```

其中：<异常对象>必须是 `Throwable` 类或其子类的对象。例如下面的语句：

```
throw new Exception("这是一个异常");
```

下面的语句在编译时将会产生语法错误：

```
throw new String("这是一个异常");
```

这是因为 `String` 类不是 `Throwable` 类的子类。

又如：

```
public void f() throws MyException{
    ...
    if( a < 0 ) throw new MyException("这是一个异常");
    ...
}
```

由于方法 `f()` 中可能会抛出 `MyException` 异常且该异常调用者应该重视它，故 `f()` 方法首先声明抛出异常 `MyException`，要求编译器对调用者使用 `f()` 方法时是否明确处理 `MyException` 异常进行强制检查。其次，当执行到 `if(a<0)` 语句时，若条件为真，则立即抛出 `MyException` 异常对象且该方法立即结束，此时该 `if()` 语句到方法 `f()` 结束之间的代码不会被执行。

【例 5.14】 从键盘读入字符，显示出其码值。若按了 `A` 键，则立即抛出异常。

ThrowExceptionTest.java

```
import java.io.*;
public class ThrowExceptionTest {
    public static void main(String[] args) {
        int c;
        try {
            while ((c = System.in.read()) != -1) {
                if (c != 'a')
                    throw new Exception("请输入字母 a!");
                System.out.println(c);
            }
        } catch (IOException e) {
            System.out.println(e);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

运行程序，输入除字母 `a` 外的任意字母。

程序运行结果:

```
S
java.lang.Exception: 请输入字母a!
```

5.7.4 自定义异常类

在程序中,有时需要定义一些异常类用于描述自身程序中的异常信息,以区分其他程序的异常信息,这时就需要自定义异常类。自定义的异常类必须是 `Throwable` 类的子类,通常是从 `Exception` 类或其子类继承的。

使用 Java 内置的异常类可以描述在编程时出现的大部分异常情况。除此之外,用户还可以自定义异常。用户自定义异常类,只需继承 `Exception` 类或其子类即可。

在程序中使用自定义异常类,大体可分为以下几个步骤。

- (1) 创建自定义异常类。
- (2) 在方法中通过 `throw` 关键字抛出异常对象。
- (3) 如果在当前抛出异常的方法中处理异常,可以使用 `try-catch` 语句捕获并处理;否则,在方法的声明处通过 `throws` 关键字指明要抛出给方法调用者的异常,继续进行下一步操作。
- (4) 由方法调用者捕获并处理异常。

【例 5.15】 使用自定义异常类,当 a 的值小于 10 或大于 100 时,将产生异常。

MyExceptionTest.java

```
/** 由于是从 Exception 类继承,故 MyException 是一个 checked exception,即必须要加以处理的异常
    否则编译程序编译通不过 */
class MyException1 extends Exception {
    int num;
    MyException1(int a) {
        num = a;
    }
    public String toString() {
        return num + "<10!\r\n 值必须大于 10";
    }
}

// MyException2 也是一个要检查的异常(checked exception)
class MyException2 extends Exception {
    int num;
    MyException2(int a) {
        num = a;
    }
    public String toString() {
        return num + ">100!\r\n 值必须小于 100";
    }
}

// 必须加以声明,因为代码内部只抛出了异常,但没有处理
class MyExceptionTest {
    static void makeException(int a) throws MyException1, MyException2 {
        if (a < 10)
```



```

        throw new MyException1(a);
    if (a > 100)
        throw new MyException2(a);
    System.out.println("没有产生例外");
}
public static void main(String args[]) {
    int a;
    try {
        a = Integer.parseInt(args[0]);
        makeException(a);
        System.out.println("a=" + a);
    } catch (MyException1 e) {
        System.out.println("产生第一个例外: \r\n" + e);
    } catch (MyException2 e) {
        System.out.println("产生第二个例外:\r\n" + e);
    }
}
}

```

右击“MyExceptionTest.java”，选择“Run As”→“Run Configurations”，如图 5.6 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_05”，在“Main class”栏中选择“MyExceptionTest”，选择“Arguments”标签页，在“Program arguments”栏中输入“9”，然后单击“Run”按钮，运行程序。

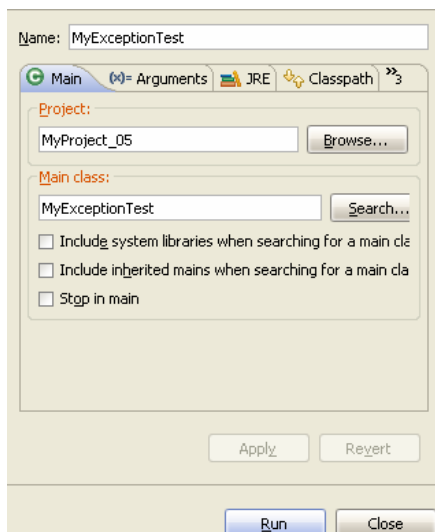


图 5.6 使用自定义异常

程序运行结果：

```

产生第一个例外:
9<10!
值必须大于10

```

5.8 综合实例

统计任意一个字符串中的英文单词总数、最长单词的长度、最短单词的长度、单词的平均长度。英文单词定义为字母串，非字母的字符都是单词之间的分隔符，例如，"ab+12cd*123fdfg%^&()as23BG"，则单词总数为 5，最长长度为 4，最短长度为 2，平均长度为 2.4。

思路：识别单词的状态转换图如图 5.7 所示。

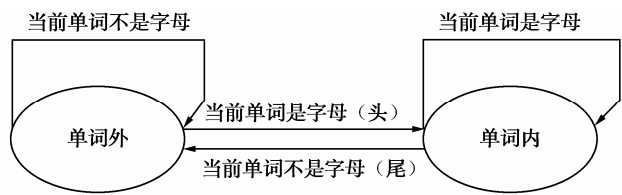


图 5.7 识别单词的状态转换图

【例 5.16】 分析字符串，计算英文单词总数、最长单词的长度、最短单词的长度、单词的平均长度。

```
public class CalculateWord {
    public static void main(String[] args) {
        String data = "ab+12cd*123fdfg%^&()as23BG";
        int count = 0; // 单词总数
        int llen = 0; // 最长单词长度
        int slen = Integer.MAX_VALUE; // 最短单词长度
        int tlen = 0; // 单词总的长度
        int wlen = 0; // 当前单词长度
        boolean inword = false; // 若为 true，表示当前状态在单词内，否则在单词外
        char ch;
        for (int i = 0; i < data.length(); i++) {
            ch = data.charAt(i);
            if (!inword) { // 判断当前是否在单词内
                if (Character.isLetter(ch)) { // 判断当前单词是否是单词
                    inword = true;
                    count++; wlen = 1;
                }
            } else {
                if (Character.isLetter(ch)) {
                    wlen++; // 当前单词长度为 1
                } else {
                    inword = false;
                    tlen += wlen; // 计算新的单词的总长度
                    if (llen < wlen)
                        llen = wlen; // 新的最长的单词长度
                    if (slen > wlen)
```

```

        slen = wlen;                                // 新的最短的单词长度
    }
}
}
// *****若是在单词内结束的，要将最后一个单词计算在内*****
if (inword) {
    tlen += wlen;
    if (llen < wlen)
        llen = wlen;
    if (slen > wlen)
        slen = wlen;
}
System.out.println("原串: " + data);
System.out.println("单词总数: " + count);
System.out.println("最长长度: " + llen);
System.out.println("最短长度: " + slen);
System.out.println("单词总长: " + tlen);
System.out.println("平均长度: " + (float) tlen / count);
}
}

```

程序运行结果:

```

原串: ab+12cd*123fdfg%^&{}as23BG
单词总数: 5
最长长度: 4
最短长度: 2
单词总长: 12
平均长度: 2.4

```

第 6 章 数组与枚举

数组是很有用的数据类型之一，是一组数据的集合，数组中的每个成员称为元素。在 Java 语言中，除了基本数据类型外，其他数据类型都是引用类型，包括数组。数组中的元素可以是任意数据类型（包括基本类型和引用类型），但同一个数组里只能存放相同类型的元素。

6.1 一维数组

1. 一维数组的定义

一维数组的定义格式有如下两种。

方式 1: <类型><数组名>[];

方式 2: <类型>[] <数组名>;

其中：<类型>可以是 Java 中任意的数据类型，<数组名>为用户自定义的一个合法的变量名，[]指明该变量是一个数组类型变量。Java 在定义数组时并不为数组元素分配内存，仅为<数组名>分配一个引用变量的空间。例如下面的语句：

```
int a[]; String [] person;  
int b[100]; // 错误，声明数组时不能指定其长度
```

2. 创建一维数组对象

和创建其他 Java 对象一样，同样使用 new 关键字创建一维数组对象，格式为：

```
数组名 = new 元素类型 [元素个数];
```

例如下面的语句：

```
int [] Array = new int[100]; // 创建一个 int 型数组，存放 100 个 int 类型的数据
```

Java 虚拟机首先在堆区中为数组分配内存空间，如图 6.1 所示，创建了一个包含 100 个元素的 int 型数组，数组成员都是 int 类型，占 4 个字节，因此整个数组对象在堆区中占用 400 个字节。之后，就要给每个数组成员赋予其数据类型的默认值，int 型的默认值是 0。

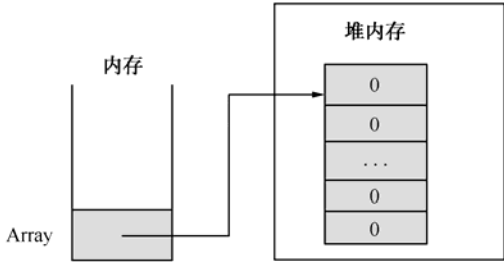


图 6.1 一维数组的内存布局

3. 一维数组初始化

定义数组的同时也可对数组元素进行显式初始化，有动态初始化和静态初始化。静态初始化指的是在定义数组的同时就为数组元素分配空间并赋值，它的格式如下。

```
<类型>[] <数组名> = {<表达式 1>,<表达式 2>,...};
```

或者

```
<类型> <数组名>[] = {<表达式 1>,<表达式 2>,...};
```

Java 编译程序会自动根据<表达式>个数算出整个数组的长度，并分配相应的空间，例如下面的语句：

```
int[] Array = {1,2,3,4};
```

数组成员是引用类型的也可静态初始化，如图 6.2 所示。

```
Point[] pa = {new Point(1,4),new Point(3,9),new Point(15,18)};
class Point {
    int x,y;
    Point(int a,int b){
        x = a;  y = b;
    }
}
```

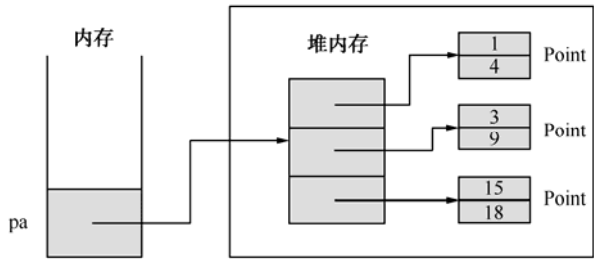


图 6.2 引用类型数组的内存布局

动态初始化指的是数组定义与为数组分配空间和赋值的操作分开进行，例如下面的语句：

```
int a[] = new int[3];
```

```
a[0] = 1; a[1] = 5; a[2] = 9;
```

同样，数组成员是引用类型的也可动态初始化，例如下面的语句：

```
Point[] pa = new Point[3];
```

```
pa[0]= new Point(1,4);
```

```
pa[1]= new Point(3,9);
```

```
pa[2]= new Point(15,18);
```

```
// 或采用匿名数组写法
```

```
// Point[] pa = {new Point(1,4),new Point(3,9),new Point(15,18)};
```

```
class Point {
    int x,y;
    Point(int a,int b){
        x = a;  y = b;
    }
}
```

【例 6.1】 输入一组非 0 整数到一维数组中，设计一个 Java 程序，求出这一组数的平均值，并分别统计出这一组数中正数和负数的个数。

TestAverage.java

```
package org.arrays;

public class TestAverage {
    public static void main(String[] args){
        int i = args.length;           // 获取命令行参数的长度
        int[] arr = new int[10];
        int num =0;
        int k =0; int p =0;
        for(int j =0;j<i;j++){
            arr[j]= Integer.parseInt(args[j]);
            if (arr[j]< 0){
                k++;                    // 负数的个数加 1
            }
            else
                p++;                    // 正数的个数加 1
            num = num + arr[j];         // 计算累加和
        }
        System.out.println("正数的个数"+p);
        System.out.println("负数的个数"+k);
        System.out.println("平均值是"+num/i);    // 计算平均值
    }
}
```

右击 “TestAverage.java”，选择 “Run As” → “Run Configurations”，如图 6.3 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_06”，在“Main class”栏中选择“Recurrence”，选择 “Arguments” 标签页，在 “Program arguments” 栏中输入 “3 8 4 -5 6 7 8 -4 11 12”，然后单击 “Run” 按钮，运行程序。

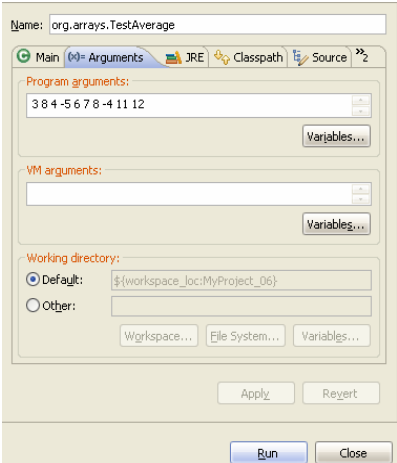


图 6.3 计算平均值

程序运行结果：

正数的个数8
负数的个数2
平均值是5

6.2 多维数组

如前所述，在 Java 语言中，多维数组实际上是数组的数组。一个二维数组可以看成一维数组，其中的每个成员又是一维数组。

1. 二维数组的定义

二维数组的定义格式如下。

格式 1: <类型>[] [] <数组名>;

格式 2: <类型> <数组名>[] [];

与一维数组的情形相类似，定义二维数组时不需要给出数组大小。

```
int a[][]; String [][] names;
```

```
int b[100][100]; // 错误，声明数组时不能指定其长度
```

2. 创建二维数组对象

和创建一维数组一样，创建二维数组同样使用 **new** 关键字，格式如下。

数组名 = new 数组元素类型 [数组元素个数] [数组元素个数];

例如下面的语句：

```
int [][] a = new int[2][3]; // 创建一个 int 型的二维数组
```

和一维数组一样，若没有对二维数组成员进行显式初始化，则会进行隐式初始化，会根据创建的数组类型初始化对象，内存布局如图 6.4 所示。

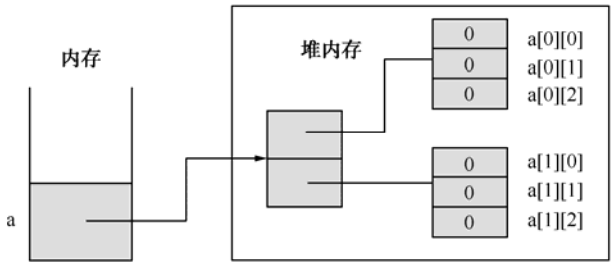


图 6.4 二维数组的内存布局

3. 二维数组初始化

和一维数组一样，定义二维数组的同时也可对数组成员进行显式初始化，有动态初始化和静态初始化。

在下面的语句中，定义 **String** 类型数组 **alphabet** 的同时初始化数组成员，即静态初始化。

```
String[] alphabet = {{ "a","b","c"}, {"d","e","f"}, {"g","h","i"}, {"j","k","l"}};
```

注意：无论初始化一维数组还是二维数组，都不能指定其长度。例如下面的语句：

```
String[4][3] alphabet = {{ "a","b","c"}, {"d","e","f"}, {"g","h","i"}, {"j","k","l"}}; // 错误
```

二维数组的第二维的长度可以各不相同，例如下面的语句：

```
String[] alphabet = {{ "a","b","c","d"}, {"e","f"}, {"g","h"}, {"i","j","k"}};
```

和一维数组一样，二维数组成员是引用类型的，也可静态初始化。下面的代码片段创建的引用类型的二维数组的内存布局如图 6.5 所示。

```
Point[][] pt = { { new Point(1, 2), new Point(3, 4)}, { new Point(5, 6),new Point(7, 8)}  
               , {new Point(9, 10),new Point(11, 12)}};  
  
class Point {  
    int x, y;  
    Point(int a, int b) {  
        x = a;    y = b;  
    }  
}
```

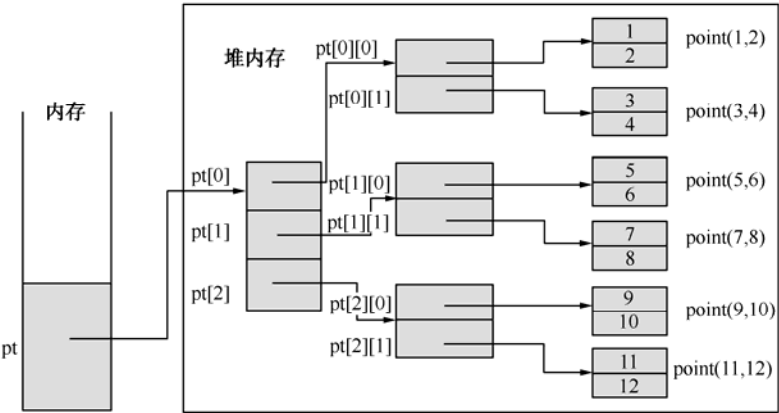


图 6.5 二维数组引用类型的内存布局

二维数组的动态初始化有以下两种方式。

方式 1:

```
<类型>[ ][ ]<数组名> = new <类型>[<第一维大小>][<第二维大小>;
```

```
例如， int[][] arr = new int[3][5];
```

方式 2: 从高维向低维依次进行空间分配，此时分配的数组空间可以是任意的形状。

(1) 由于二维数组首先是一个一维数组，故先对该一维数组进行空间分配，也就是说，先对最高维进行空间分配。

```
<类型>[ ][ ]<数组名> = new <类型>[<元素个数>][];
```

```
例如， int [][] c = new int[3][];
```

(2) 每一个元素又是一个一维数组，故对每一个元素再用 new 语句进行空间分配，格式与一维数组相同。例如下面的语句:

```
c[0] = new int[4];
```

```
c[1] = new int[3];
```

```
c[2] = new int[5];
```

(3) 若最终元素是引用类型，则还需对每一个最终元素进行对象的空间分配。例如下面的语句:

```
Point[][] p;           // 定义一个二维数组的引用变量  
p = new Point[3][];    // 先作为一维数组，进行最高维空间的分配  
p[0] = new Point[2];   // 每一个元素又是一个一维数组，再进行一维数组空间分配  
p[1] = new Point[1];  
p[2] = new Point[2];  
p[0][0] = new Point(1,1);
```



```

p[0][1] = new Point(2,2);           // 最后对每一个元素进行 Point 对象空间的分配
p[1][0] = new Point(3,3);
p[2][0] = new Point(4,4);
p[2][1] = new Point(5,5);
class Point {
    int x, y;
    Point(int a, int b) {
        x = a;   y = b;
    }
}

```

【例 6.2】 设计一个 Java 程序，从小到大将从命令行中读取的一组数字进行升序排列。
NumSort.java

```

package org.arrays;
public class NumSort {
    public static void main(String[] args) {
        int[] a = new int[args.length];
        for (int i=0; i<args.length; i++) {           // 获取命令行参数的长度
            a[i] = Integer.parseInt(args[i]);         // 把字符串转换为整型
        }
        System.out.println("排序前: ");
        print(a);
        selectionSort(a);
        System.out.println("排序后: ");
        print(a);
    }
    private static void selectionSort(int[] a) {       // 排序
        int k, temp;
        for(int i=0; i<a.length; i++) {
            k = i;
            for(int j=k+1; j<a.length; j++) {
                if(a[j] < a[k]) {
                    k = j;
                }
            }
            if(k != i) {
                temp = a[i];  a[i] = a[k];  a[k] = temp;
            }
        }
    }
    private static void print(int[] a) {
        for(int i=0; i<a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }
}

```

右击“NumSort.java”，选择“Run As”→“Run Configurations”，如图 6.6 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_06”，在“Main class”栏中选择“NumSort”，选择“Arguments”标签页，在“Program arguments”栏中输入“12 5 78 45 17 6 8 13 32 14”，然后单击“Run”按钮，运行程序。

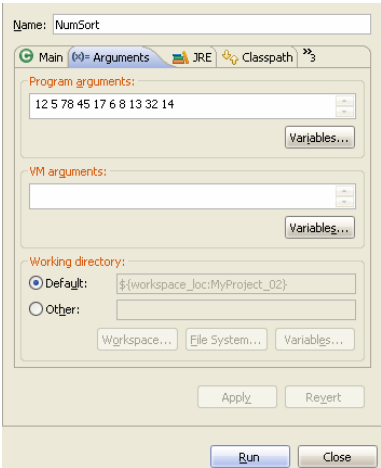


图 6.6 对数值按从小到大顺序排列

程序运行结果：

```
排序前：
12 5 78 45 17 6 8 13 32 14
排序后：
5 6 8 12 13 14 17 32 45 78
```

6.3 访问数组

数组中的每个元素都有一个索引，或者称为下标。数组中的第一个元素的索引为 0，第二个元素的索引为 1，依次类推。数组的 length 属性表示数组的长度，它的声明形式如下。

```
public final int length;
```

length 属性被 public 和 final 修饰，因此在程序中可以读取数组的长度，但不能修改数组的长度。

【例 6.3】 求出 4 阶矩阵的最大元素以及其所在的行号和列号。

MatMax.java

```
package org.arrays;

public class MatMax {
    public static void main(String[] args) {
        int arr[][] = {{ 12,76,4,1},{ 19,28,55,6},{ 2,10,13,2},{ 3,9,110,22}};
        int i =0; int j=0 ;int row=0;int col=0;int max=arr[0][0];
        System.out.println("4 阶矩阵是： ");
        for(i=0;i<4;i++){
            for(j=0;j<4;j++){
                System.out.print(arr[i][j]+"\\t");
                if(max<arr[i][j]){
```

```

        max = arr[i][j];
        row = i ; col = j;
    }
}
System.out.println();
}
System.out.println("矩阵的最大数是"+max);
System.out.println("该数位于矩阵的第"+row+"列"+"第"+col+"行");
}
}

```

程序运行结果:

```

4阶矩阵是:
12      76      4      1
19      28      55     6
2       10      13     2
3       9       110    22
矩阵的最大数是110
该数位于矩阵的第3列第2行

```

【例 6.4】 将一个 3×4 阶矩阵转置。矩阵转置就是将一个矩阵的行、列互换。

MatInverse.java

```

package org.arrays;

public class MatInverse {
    public static void main(String[] args) {
        int a[][]= {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
        int b[][] = new int[4][3];
        int i =0 ; int j =0;
        System.out.println("矩阵转置前: ");
        for (i=0;i<3;i++){
            for(j=0;j<4;j++){
                System.out.print(a[i][j]+"");
            }
            System.out.println();
        }
        for(i=0;i<3;i++){
            for(j=0;j<4;j++){
                b[j][i]= a[i][j];
            }
        }
        System.out.println("矩阵转置后: ");
        for (i=0;i<4;i++){
            for(j=0;j<3;j++){
                System.out.print(b[i][j]+"");
            }
            System.out.println();
        }
    }
}

```

程序运行结果：

矩阵转置前:			
1	2	3	4
5	6	7	8
9	10	11	12
矩阵转置后:			
1	5	9	
2	6	10	
3	7	11	
4	8	12	

6.4 数组实用类：Arrays

在 `java.util` 包中，有一个用于操作数组的实用类：`java.util.Arrays`，它提供了一系列的静态方法。

6.4.1 复制数组

`java.lang.System` 类提供了一个很有用的静态方法 `arraycopy()` 用来复制数组，其语法格式如下。

```
static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

功能：从指定源数组中复制一个数组，复制从指定的位置开始，复制到目标数组的指定位置。从 `src` 引用的源数组到 `dest` 引用的目标数组，数组组件的一个子序列被复制下来。被复制的组件的数量等于 `length` 参数。源数组中位置在 `srcPos~srcPos+length-1` 之间的组件被分别复制到目标数组中的 `destPos~destPos+length-1` 位置。

【例 6.5】 使用 `arraycopy()` 方法把数组的内容复制到另一个数组。

CopyingArrays.java

```
package org.arrays;
import java.util.Arrays;
public class CopyingArrays {
    public static void main(String[] args) {
        int[] i1 = new int[] { 6, 18, 24 };
        int[] i2 = new int[10];
        System.arraycopy(i1, 1, i2, 5, 2); // 复制数组的元素
        for (int i = 0; i < i2.length; i++) {
            System.out.print(i2[i]+" ");
        }
        System.out.println();
        Integer[] array1 = { 3, 5, 9, 15 };
        Integer[] array2 = new Integer[10];
        System.arraycopy(array1, 1, array2, 5, 3);
        for (int i = 0; i < array2.length; i++) {
            System.out.print(array2[i] + " ");
        }
        System.out.println();
        System.out.print(Arrays.deepToString(array2) + " "); // 显示数组的元素
        System.out.println();
    }
}
```

```

        Person[] p1 = new Person[] { new Person("Tom", 18),
                                      new Person("Jack", 20), new Person("Lucy", 24) };
        Person[] p2 = new Person[3];
        System.arraycopy(p1, 0, p2, 0, p1.length);
        for (int i = 0; i < p2.length; i++) {
            System.out.println("name=" + p2[i].name + "," + "age=" + p2[i].age);
        }
        System.out.println("name=" + p1[1].name + "," + "age=" + p1[1].age);
        System.out.print(Arrays.deepToString(p1) + " ");
    }
}

class Person {
    String name;    int age;
    Person(String x, int y) {
        this.name = x;  this.age = y;
    }
    public String toString() {
        return "name=" + name + "," + "age=" + age;
    }
}

```

程序运行结果：

```

0 0 0 0 0 18 24 0 0 0
null null null null null 5 9 15 null null
[null, null, null, null, null, 5, 9, 15, null, null]
name=Tom,age=18
name=Jack,age=20
name=Lucy,age=24
name=Jack,age=20
[name=Tom,age=18, name=Jack,age=20, name=Lucy,age=24]

```

6.4.2 数组排序

使用内置的排序方法，就可以对任意基本类型数组排序，也可以对任意的对象数组进行排序，只要该对象实现了 `Comparable` 接口或具有相关联的 `Comparator`。当然，可以对数组按指定的排序方法进行排序。

【例 6.6】 随机生成 10 个整数值，并使用内置的排序方法对数组排序。

ArraysSort.java

```

package org.arrays;
import java.util.Arrays;
import java.util.Random;
public class ArraysSort {
    public static void main(String[] args){
        Random r = new Random();           // 创建 Random 对象，用于生成伪随机数
        int[] a = new int[10];
        System.out.println("排序前: ");
        for (int i=0;i<10;i++){
            a[i] = r.nextInt(100);          // 获取在 0~100 之间的整型随机数
        }
    }
}

```

```

        System.out.print(a[i]+" ");
    }
    System.out.println();
    Arrays.sort(a);                // 从小到大对数组进行排序
    System.out.println("排序后: ");
    for (int i=0;i<10;i++){
        System.out.print(a[i]+" ");
    }
}
}

```

程序运行结果:

```

排序前:
35 59 92 70 9 52 44 2 96 78
排序后:
2 9 35 44 52 59 70 78 92 96

```

6.4.3 数组元素的查找

如果数组已经排好序了, 就可以使用 `Array.binarySearch()` 执行快速查找。

【例 6.7】 对无序的 10 个整数值进行排序, 再用二叉查找法进行检索。

ArraySearch.java

```

package org.arrays;
import java.util.Scanner;
import java.util.Arrays;
public class ArraySearch {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 从控制台获取要检索的值
        int[] array = { 23, 4, 6, 21, 60, 99, 42, 69, 53, 36 };
        System.out.println("排序前: ");
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");
        System.out.println();
        Arrays.sort(array);                // 数组排序
        System.out.println("排序后: ");
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");
        System.out.println("\n 请输入查找值: ");
        int key = scanner.nextInt();
        int find = -1;
        if ((find = Arrays.binarySearch(array, key)) > -1) {
            System.out.println("找到值位于索引 " + find + " 位置");
        } else
            System.out.println("找不到指定值");
    }
}

```

程序运行结果：

```
排序前：
23 4 6 21 60 99 42 69 53 36
排序后：
4 6 21 23 36 42 53 60 69 99
请输入查找值：
42
找到值于索引 5 位置
```

6.5 枚举

JDK5.0 引入了一个新的关键字 `enum` 表示枚举类型。定义一个枚举类型很简单，下面是一个枚举类型的示例。

```
public enum Season {
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

上面创建了一个名为 `Season` 的枚举类型，它具有 4 个成员。由于枚举类型的实例是常量，因此按照命名习惯它们都用大写字母表示。为了使用 `enum`，需要创建一个该类型的引用，将其赋值给某个实例。例如下面的语句：

```
Season season = Season. SUMMER;
```

在创建一个 `enum` 的实例后，编译器会自动创建一些有用的方法。`ordinal()` 方法用来表示某个特定 `enum` 常量的声明顺序，`values()` 方法用来按照 `enum` 常量的声明顺序，产生由这些常量值构成的数组。例如下面的代码：

```
for (Season season : Season.values()) {
    System.out.println(season+"",ordinal "+season.ordinal());
}
```

输出：

```
SPRING, ordinal 0
SUMMER, ordinal 1
AUTUMN, ordinal 2
WINTER, ordinal 3
```

定义枚举类型本质上就是在定义一个 `final` 类型的 `class`，该类从 `java.lang.Enum` 类继承。这些工作由编译器来完成，所以 `enum` 像类一样可以具有自己的成员方法、数据成员、自己的构造方法、自己的初始化代码块，以及内部类。

【例 6.8】 用 `enum` 模拟交通信号灯。

TrafficLight.java

```
package org.enums;
enum Signal {
    GREEN, YELLOW, RED;
}
public class TrafficLight {
    Signal color = Signal.RED;
    public void change() {
        switch (color) {
            case RED:color = Signal.GREEN;           // 注意：在 case 中不能写成 Signal.RED
```

```

        break;
        case GREEN:color = Signal.YELLOW; break;
        case YELLOW:color = Signal.RED; break;
    }
}
public String toString(){
    return "The traffic light is " + color;
}
public static void main(String[] args){
    TrafficLight t = new TrafficLight();
    for(int i = 0;i< 7;i++){
        System.out.println(t);
        t.change();
    }
}
}

```

程序运行结果:

```

The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED

```

6.6 enum的构造方法

使用 enum 定义枚举类型时，实质上定义出来的类继承自 `java.lang.Enum` 类，而每个枚举的成员其实就是定义的枚举类型的一个实例。在定义一个枚举类型时，像定义类一样，可以定义枚举类型的构造方法，这样在定义枚举类型的成员变量时，利用构造方法进行初始化。在定义枚举类型时，必须将枚举常量定义在最前面，并以分号“;”与其他成员隔开。若 enum 是 public 类型，且在类外部定义，则文件名必须与 enum 名字相同，且文件中不能再定义其他 public 类型的类。

注意：enum 的构造方法必须是 private，否则出错。

【例 6.9】 在 enum 中定义 enum 的构造方法和普通方法。

Orientation.java

```

package org.enums;
public enum Orientation {
    EAST("shanghai"),SOUTH("shenzhen"),WEST("xian"),NORTH("beijing");
    private String city;
    Orientation(String city) {           // enum 的构造方法，编译程序自动加上 private 修饰符
        this.city = city;
    }
    public String getCity() {           // enum 的普通方法
        return city;
    }
    public static void main(String[] args) {
        Orientation or1 = Orientation.EAST;
    }
}

```



```

        Orientation or2 = Orientation.SOUTH;
        Orientation or3 = Orientation.WEST;
        Orientation or4 = Orientation.NORTH;
        System.out.println(or1.getCity());
        System.out.println(or2.getCity());
        System.out.println(or3.getCity());
        System.out.println(or4.getCity());
    }
}

```

程序运行结果：

```

shanghai
shenzhen
xian
beijing

```

注意：可以由枚举对象的名字串，构造一个相应的枚举对象，即由 String 转换成枚举对象。这一转换也是经常使用的，例如，由名字串“EAST”转换成枚举对象 EAST，代码如下。

```
Orientation or1 = Orientation.valueOf("EAST");
```

此时，or1 与 Orientation.EAST 是同一个对象。

6.7 综合实例

【例 6.10】 设计一个 Java 程序，打印 n 阶螺旋矩阵。例如当 $n=3$ 时，则：

```

1   2   3
8   9   4
7   6   5

```

思路：数据的存储采用二维矩阵，数据存储的方向使用枚举常量左、右、上、下来表示。

Test.java

```

package org.arrays;

public class Test {
    int[][] arr = new int[4][4];           // 初始化矩阵的行和列
    int i = 0;                             // 横坐标
    int j = 0;                             // 纵坐标
    int num = 1;                           // 初始值为 1
    public void spire() {
        if (arr.length != 1) {
            Direction dir = Direction.RIGHT; // 初始方向
            while (arr[i][j] == 0) {
                arr[i][j] = num++;
                switch (dir) {                // 判断当前方向以及转向
                    case RIGHT:
                        if ((j + 1 < arr[0].length) && (arr[i][j + 1] == 0)) {
                            j++;
                        } else {

```

```

        i++;
        // 转入矩阵的下一行
        dir = Direction.DOWN;
    }
    break;
case DOWN:
    if ((i + 1 < arr.length) && (arr[i + 1][j] == 0)) {
        i++;
    } else {
        j--;
        dir = Direction.LEFT;
        // 转入矩阵的左方位
    }
    break;
case LEFT:
    if ((j - 1 >= 0) && (arr[i][j - 1] == 0)) {
        j--;
    } else {
        i--;
        dir = Direction.UP;
        // 转入矩阵的上方位
    }
    break;
case UP:
    if ((i - 1 >= 0) && (arr[i - 1][j] == 0)) {
        i--;
    } else {
        j++;
        dir = Direction.RIGHT;
        // 转入矩阵的右方位
    }
    break;
    }
    }
    print(arr);
    // 打印
} else
    System.out.println(1);
}
// 打印矩阵数组
public static void print(int[][] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[0].length; j++) {
            System.out.print(arr[i][j] + "\t");
        }
        System.out.println();
    }
}
// 用来表示方向的枚举
public enum Direction {
    RIGHT, DOWN, LEFT, UP
}

```

```

    public static void main(String[] args) {
        Test t = new Test();
        t.spire();
    }
}
// 计算螺旋矩阵各元素的值

```

程序运行结果：

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

【例 6.11】 找二维数组中的马鞍点。

马鞍点为本行中最小，但本列中最大的点。假设二维数组中没有重复数据。

思路：首先从二维数组的第一行寻找每一行的最小值，找到后，记下该行的最小值在数组中的列号，由这个列号再判断它是否是本列中的最大的点。如果是，则找到了马鞍点。如果不是，则继续按照上述方法查找，直到找到马鞍点。

SaddlePoint.java

```

package org.arrays;
public class SaddlePoint {
    public static void main(String[] args) {
        int[][] a = { { 1, 3, 4, 7 }, { 5, 6, 2, 0 }, { 16, 36, 61, 18 } };
        int lineMin;
        int minx;
        int miny;
        again: for (int i = 0; i < a.length; i++) {
            lineMin = a[i][0];
            minx = i; miny = 0;
            //*****找第 i 行中最小值*****
            for (int j = 0; j < a[i].length; j++) {
                if (a[i][j] < lineMin) {
                    lineMin = a[i][j];
                    minx = i; miny = j;
                }
            }
            //*****判断该最小值是否是列中最大*****
            for (int j = 0; j < a.length; j++) {
                if (a[j][miny] > lineMin)
                    continue again;
            }
            System.out.println("马鞍点: " + lineMin + " 位于: (" + minx + ", " + miny + ")");
        }
    }
}

```

程序运行结果：

马鞍点: 16 位于: { 2 , 0 }

第 7 章 容器和泛型

Java 有多种方式保存对象，如在第 6 章学习的数组。数组是保存一组对象的最有效方式，如果想保存一组基本数据类型，适合用数组来保存。但是数组一旦创建，其长度就不能改变。而在写程序时可能并不知道将需要多少个对象，或者需要更复杂且更有效的方式来存储对象，因此使用数组来保存对象就不适宜了。

Java 实用类库提供了一套相当完整的容器类来解决这个问题，其中基本的类型是 List、Set、Queue 和 Map，这些对象类型也称为集合类。容器提供了完善的方法来保存对象。Java 的容器类库位于 java.util 包中，Java 容器类库中的接口及类之间的关系如图 7.1 所示，图中实线表示继承，虚线表示接口实现。而泛型的引入最主要原因就是安全地使用容器类。

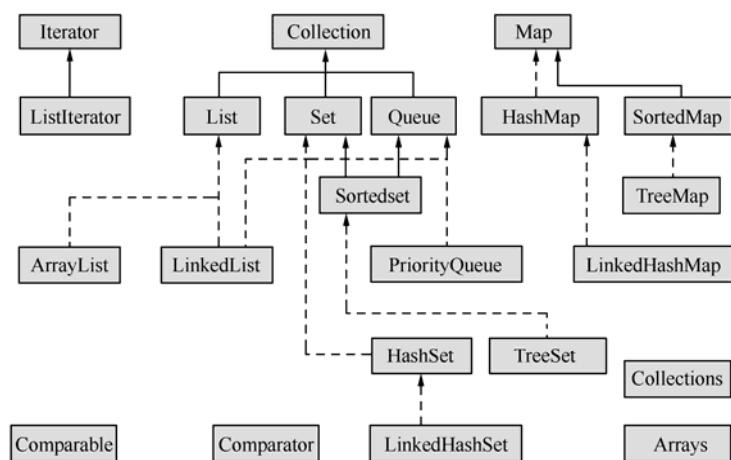


图 7.1 容器类

7.1 Collection与Iterator

Collection 是容器类的根接口，**List**、**Set**、**Queue** 是它的直接子接口。**Collection** 表示一组对象，这些对象也称为 **Collection** 的元素。**List** 类型的容器允许加入重复对象，按照索引位置排序并且按照在容器中的索引位置检索对象。**Set** 类型的容器不允许加入重复对象，也不按照某种方式排序对象。**Map** 接口没有继承 **Collection** 接口。**Map** 类型的容器中的每一个元素包含一对键对象和值对象，容器中的键对象不可重复，值对象可以重复。表 7.1 列出了 **Collection** 接口的常用方法。

表 7.1 Collection 接口的常用方法

方 法	描 述
boolean add(E e)	向容器中添加一个元素

方 法	描 述
boolean addAll(Collection<? extends E> c)	向容器中添加参数中所有的元素
void clear()	移除容器中的所有元素
boolean contains(Object o)	判定此 collection 是否包含指定的元素，有则返回 true
boolean containsAll(Collection<?> c)	判定此 collection 是否包含指定 collection 中的所有元素，是则返回 true
boolean isEmpty()	判定此容器是否为空，是则返回 true
Iterator<E> iterator()	返回一个 Iterator<T>，用来遍历容器中的所有元素
boolean remove(Object o)	如果容器中存在此元素，则删除它
boolean retainAll(Collection<?> c)	将此 Collection 与参数 c 的交集存入此 Collection 中
int size()	返回此 collection 中的元素数目
Object[] toArray()	返回包含此 collection 中所有元素的数组

Collection 接口的 iterator()和 toArray()方法都用于获得容器中的所有元素，前者返回一个 Iterator 对象，后者返回一个包含容器中所有元素的数组。

Iterator 接口中声明了如下方法。

- boolean hasNext(): 判断容器中的元素是否遍历完毕，没有则返回 true。
- next(): 返回迭代的下一个元素。
- void remove(): 从迭代器指向的 Collection 中移除迭代器返回的最后一个元素。必须先调用一次 next()方法之后，才能调用一次 remove()，即 remove()不能连续多次调用。

【例 7.1】 向容器中添加一组元素，用 iterator()方法遍历容器中的元素。

UseCollection.java

```
package org.container;
import java.util.*;
public class UseCollection {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<String>(Arrays.asList(
            "A", "B", "C", "D", "E")); // 创建容器
        String[]strArray = { "F", "G", "H", "I", "J" };
        collection.addAll(Arrays.asList(strArray));
        Collections.addAll(collection, "M", "N", "O", "P", "Q");
        System.out.println(collection);
        //迭代容器中的每一个元素
        for(Iterator it = collection.iterator();it.hasNext();){
            System.out.print(it.next() + " ");
        }
        System.out.println();
        for (String i : collection) {
            System.out.print(i + " ");
        }
        System.out.println();
        Collection.remove("A"); // 移除一个元素
        for (String i : collection) {
```

```

        System.out.print(i + " ");
    }
    Collection<String> part = new ArrayList<String>(Arrays.asList(
        "B", "C", "D", "E"));
    System.out.println();
    collection.retainAll(part);                // 保存相同的元素
    for (String str : collection) {
        System.out.print(str + " ");
    }
    System.out.println();
    Object[] o = collection.toArray();        // 返回一个数组
    System.out.println(Arrays.deepToString(o));
    collection.removeAll(collection);         // 移除所有的元素
    System.out.println(collection.size());
}
}

```

说明：在程序中使用 `Arrays.asList()` 方法接收一个用逗号分隔的元素列表，并将其转换为 `ArrayList` 对象。`Collections.addAll()` 方法接收一个 `Collection` 对象，以及一个用逗号分隔的列表，将元素添加到 `Collection` 中。

程序运行结果：

```

[A, B, C, D, E, F, G, H, I, J, M, N, O, P, Q]
A B C D E F G H I J M N O P Q
A B C D E F G H I J M N O P Q
B C D E F G H I J M N O P Q
B C D E
[B, C, D, E]
0

```

7.2 实用类Collections

就像 `java.util.Arrays` 一样，`java.util.Collections` 类也有一些实用的 `static` 方法，其中一部分方法专门用于操纵 `List` 类型容器，还有一部分方法可用于操纵所有的 `Collection` 类型或 `Map` 类型容器。

`List` 代表长度可变的线性表，`Collections` 的以下方法适用于 `List` 类型。

- `copy(List<? super T> dest, List<? extends T> src)`：将所有元素从一个列表复制到另一个列表。

- `fill(List<? super T> list, T obj)`：使用指定元素替换指定列表中的所有元素。

- `nCopies(int n, T o)`：返回由指定对象的 n 个副本组成的不可修改的列表。

- `shuffle(List<?> list)`：使用默认随机源对指定列表进行置换。

- `sort(List<T> list)`：根据元素的自然顺序对指定列表按升序进行排序。

【例 7.2】 使用 `Collections` 的 `min()`、`max()`、`binarySearch()` 等常用方法。

`CollectionsOfList.java`

```

package org.container;
import java.util.*;

```

```

class StringAddress {
    private String s;
    public StringAddress(String s) {
        this.s = s;
    }
    public String toString() {
        return super.toString() + " " + s;
    }
}

public class CollectionsOfList {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList(new String[] { "spring", "summer", "autumn", "winter" });
        Collections.sort(list1); // 把 List 中的元素自然排序
        System.out.println(Collections.max(list1)); // 显示 winter
        System.out.println(Collections.min(list1)); // 显示 autumn
        System.out.println(Collections.binarySearch(list1, "spring")); // 显示 1
        System.out.println(Arrays.toString(list1.toArray()));
        Collections.shuffle(list1); // 重新随机调整 List 中元素的位置
        System.out.println(Arrays.toString(list1.toArray())); // 随机显示
    }
}

```

程序运行结果：

```

winter
autumn
1
[autumn, spring, summer, winter]
[winter, summer, spring, autumn]

```

7.3 Set (集)

Set 不接受重复的元素，如果试图将相同对象的多个实例添加到 Set 中，那么它就会阻止这种重复现象。所谓相同对象，就是若 `e1.equals(e2)`，就称 `e1` 与 `e2` 是重复元素、相同对象。Set 中最常被使用的是测试归属属性，可以很容易地检测某个对象是否在某个 Set 中。正因如此，查找就成了 Set 中最重要的操作。因此通常都会选择一个 HashSet 的实现，它专门对快速查找进行了优化。Set 具有与 Collection 完全一样的接口，实际上 Set 就是 Collection，只是行为不同。Set 的子接口是 SortedSet，实现类有 HashSet、TreeSet 与 LinkedHashSet。

7.3.1 HashSet

HashSet 类按照哈希算法来存取容器中的对象，具有很好的存取和查找性能。当向容器中加入一个对象时，HashSet 会调用对象的 `hashCode()` 方法来获取哈希码，然后根据这个哈希码进一步计算出对象在容器中的存放位置。

在 Object 类中定义 `hashCode()` 和 `equals()` 方法，Object 类的 `equals()` 方法按照对象的内存地址比较对象是否相等，因此如果 `object1.equals(object2)` 为 true，则表明 `object1` 变量和 `object2` 变量实际上引用同一个对象，那么 `object1` 和 `object2` 的哈希码也肯定相同。

为了保证 HashSet 能正常工作，要求当两个对象用 `equals()` 方法比较的结果为 true 时，它

们的哈希码也相等。例如，如果 `object1.equals(object2)` 为 `true`，那么以下表达式的结果也应为 `true`。

```
object1.hashCode() == object2.hashCode();
```

如果用户定义类覆盖 `Object` 类 `equals()` 方法，但没有覆盖 `Object` 类 `hashCode()` 方法，就会导致 `object1.equals(object2)` 为 `true` 时，而 `object1` 和 `object2` 的哈希码不一定一样，这会使 `HashSet` 无法正常工作。

【例 7.3】 测试不同时重载 `Object` 类的 `equals()` 和 `hashCode()` 方法。

HashSetTest.java

```
package org.container;
import java.util.*;
class Person {
    private String name;
    private int age;
    public Person(String n, int a) {
        this.name = n; this.age = a;
    }
    public boolean equals(Object obj) {
        if (obj instanceof Person) {
            Person p = (Person) obj;
            return (name.equals(p.name)) && (age == p.age);
        }
        return super.equals(obj);
    }
    // public int hashCode() {
    //     int i;
    //     i = (name == null ? 0 : name.hashCode());
    //     i = 10 * i + age;
    //     return i;
    // }
}
public class HashSetTest {
    public static void main(String[] args) {
        Collection c = new HashSet();
        c.add(new Person("Jack", 20));
        c.add(new Person("Jack", 20));
        System.out.println(c.size());
        System.out.println(c);
        c.add(new Integer(25));
        c.add("Hello World");
        c.remove(new Integer(25));
        c.remove("Hello World");
        System.out.println(c.remove(new Person("Jack", 20)));
    }
}
```


程序运行结果：

```
2
[org.container.Person@1fb8ee3, org.container.Person@c17164]
false
```

说明：在程序 HashSetTest.java 中加入两个相同的对象到 Set 中，由于并没有覆盖 Object 类 hashCode()方法，创建的两个 Person 对象的哈希码不一样，因此 HashSet 为两个 Person 对象计算不同的存放位置，于是把它们存放在容器的不同地方。可见，为了保证 HashSet 正常工作，如果 Student 类覆盖了 equals()方法，也应该覆盖 hashCode()方法，并且保证两个相等的 Person 对象的哈希码也一样。因此取消程序的注释，重新运行 HashSetTest.java 程序。程序运行结果：

```
1
[org.container.Person@15f180a]
true
```

7.3.2 TreeSet

TreeSet 类实现了 SortedSet 接口，能够对容器中的对象进行排序。当向 TreeSet 中加入一个对象后，会继续保持对象间的排序的次序。例如下面的代码片段：

```
Set set = new TreeSet();
set.add(new String("spring"));
set.add(new String("summer"));
set.add(new String("autumn"));
set.add(new String("winter"));
System.out.println(set);
```

运行结果：

```
[autumn, spring, summer, winter]
```

TreeSet 支持两种排序方式：自然排序和指定排序。在默认的情况下，TreeSet 采用自然排序方式。

1. 自然排序

在 JDK 类库中，有一部分类实现了 java.lang.Comparable 接口，如 Integer、Double 和 String 等。Comparable 接口有一个 compareTo(Object o)方法，它返回整数类型。

x.compareTo(y)：如果返回值为 0，则表示 x 和 y 相等；如果返回值大于 0，则表示 x 大于 y；如果返回值小于 0，则表示 x 小于 y。

TreeSet 调用对象的 compareTo()方法比较容器中对象的大小，然后进行升序排列。表 7.2 列出了 JDK 类库中实现了 Comparable 接口的一些类的排序及其排序方式。

表 7.2 类的排序

类	排 序 方 式
BigDecimal、Integer、Byte、Double Float、Integer、Long 、Short	按数字大小排序
Character	按字符的 Unicode 值的数字大小排序

使用自然排序时，只能向 `TreeSet` 容器中加入同类型的对象，如下所示。

```
Set set = new TreeSet();
set.add(new Integer(1));
set.add(new String("spring"));
System.out.println(set);
```

//抛出 java.lang.ClassCastException 异常

向 `TreeSet` 容器中加入同类型的对象，要求这些对象的类必须实现 `Comparable` 接口。

【例 7.4】 向 `TreeSet` 容器中加入 4 个雇员信息，并按工资的多少进行升序排列。

Employee.java

```
package org.container;
import java.util.*;

class Employee implements Comparable {
    private String name;    private int salary;
    public Employee(String name, int age) {
        this.name = name;    this.salary = age;
    }
    public String toString() {
        return "name=" + name + ", " + "salary=" + salary;
    }
    public boolean equals(Object o) {                // 重写 equals()方法
        if (!(o instanceof Employee))
            return false;
        Employee employee = (Employee) o;
        if (this.name.equals(employee.name) && this.salary == employee.salary)
            return true;
        else
            return false;
    }
    public int hashCode() {                          // 重写 hashCode()方法
        int result;
        result = (name == null ? 0 : name.hashCode());
        return result;
    }
    public int compareTo(Object o) {                 // 重写 compareTo()方法
        Employee e = (Employee) o;
        int result = salary > e.salary ? 1 : (salary == e.salary ? 0 : -1);
        if (0 == result) {
            result = name.compareTo(e.name);
        }
        return result;
    }
    public static void main(String[] args) {
        Set<Employee> set = new TreeSet<Employee>();
        set.add(new Employee("Lucy", 2800));
        set.add(new Employee("John", 4000));
        set.add(new Employee("Mary", 3000));
```

```

        set.add(new Employee("Lily", 3000));
        Iterator<Employee> it = set.iterator();
        while (it.hasNext()) {
            Employee student = it.next();
            System.out.println(student);
        }
    }
}

```

说明：为了保证 TreeSet 能正确地排序，Employee 类必须实现 Comparable 接口，其 compareTo()方法与 equals()方法按相同的规则比较两个对象是否相等。

程序运行结果：

```

name=Lucy, salary=2800
name=Lily, salary=3000
name=Mary, salary=3000
name=John, salary=4000

```

2. 指定排序

Java.util.Comparator<Type>接口提供具体的排序方式，<Type>指定被比较的对象的类型，Comparator 接口的 compare(T o1,T o2)方法用于比较两个对象的大小。当 compare(T o1,T o2)的返回值大于 0 时，表示 o1 大于 o2；当 compare(T o1,T o2)的返回值等于 0 时，表示 o1 等于 o2；当 compare(T o1,T o2)的返回值小于 0 时，表示 o1 小于 o2。

【例 7.5】 实现 Comparator 接口，加入 TreeSet 容器中的对象以 brand 降序排列，以 place 升序排列。

ComputerComparator.java

```

package org.container;
import java.util.*;

public class ComputerComparator implements Comparator<Computer> {
    public int compare(Computer c1, Computer c2) {
        if (c1.brand.compareTo(c2.brand) > 0)                // brand 按降序排列
            return -1;
        else if (c1.brand.compareTo(c2.brand) < 0)
            return 1;
        else
            return c1.place.compareTo(c2.place);            // place 按升序排列
    }

    public static void main(String[] args) {
        Set<Computer> set = new TreeSet<Computer>(new ComputerComparator());
        Computer computer1 = new Computer("dell", "Shanghai");
        Computer computer2 = new Computer("hp", "Shenzhen");
        Computer computer3 = new Computer("hp", "Guangzhou");
        Computer computer4 = new Computer("lenovo", "Beijing");
        Computer computer5 = new Computer("dell", "Beijing");
        set.add(computer1);                                // 向容器中加入元素
    }
}

```

```

        set.add(computer2);    set.add(computer3);
        set.add(computer4);    set.add(computer5);
        Iterator<Computer> it = set.iterator();           // 返回一个迭代器
        while (it.hasNext()) {
            Computer computer = it.next();
            System.out.println(computer);
        }
    }
}

class Computer {
    String brand;    String place;
    public Computer(String b, String p) {
        this.brand = b;    this.place = p;
    }
    public String toString() {
        return "computer brand: " + brand + ",    place:" + place;
    }
    public boolean equals(Object o) {                     // 重写 equals()
        if (!(o instanceof Computer))
            return false;
        Computer c = (Computer) o;
        if (this.brand.equals(c.brand) && this.place == c.place)
            return true;
        else
            return false;
    }
    public int hashCode() {                               // 重写 hashCode()
        int result;
        result = (brand == null ? 0 : brand.hashCode());
        return result;
    }
}

```

程序运行结果:

```

computer brand: lenovo,    place:Beijing
computer brand: hp,    place:Guangzhou
computer brand: hp,    place:Shenzhen
computer brand: dell,    place:Beijing
computer brand: dell,    place:Shanghai

```

7.4 List (列表)

像数组一样，List 也能建立数字索引与对象的关联，表达的是数据结构中线性表的概念。List 的主要特征是其元素以线性方式存储，容器中允许存放重复对象。List 接口的常用实现类是：ArrayList 和 LinkedList。

7.4.1 ArrayList

ArrayList 代表长度可变的数组，允许对元素进行随机的快速访问，但是向 ArrayList 中插入与删除元素的速度较慢。ArrayList 是线程不安全的，若要成为线程安全的，可用：

```
List list = Collections.synchronizedList(new ArrayList());
```

【例 7.6】 运用 ArrayList 类的各种方法，并展示相似方法的异同点。

ArrayListTest.java

```
package org.container;
import java.util.*;
public class ArrayListTest {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add(new String("rat"));           // 向列表中加入元素
        list.add(new String("monkey"));
        list.add(new String("pig"));
        list.add(new String("rabbit"));
        System.out.println("1:" + list);
        String str = new String("horse");
        list.add(str);
        System.out.println("2:" + list);
        System.out.println("3:" + list.contains(str)); // 判断是否包含该元素
        String str3 = new String("goat");
        System.out.println("4:" + list.indexOf(str3)); // 返回列表中首次出现该元素的索引
        System.out.println("5:" + list.remove(str3)); // 移除列表中首次出现的元素
        List<String> sub = list.subList(1, 4);       // 返回列表中从 1 到 3 之间的元素
        System.out.println("subList:" + sub);
        System.out.println("6:" + list.isEmpty());  // 判断列表是否为空
        System.out.println("7:" + list);
        Object[] obj = list.toArray();              // 返回一个数组，该数组包含该列表中的所有元素
        System.out.println("8:" + obj[0]);
        /** 返回一个数组，该数组包含该表的所有元素，返回数组的运行时类型与参数数组的类
            型完全相同 */
        String[] str4 = list.toArray(new String[0]);
        System.out.println("9:" + str4[1]);
    }
}
```

程序运行结果：

```
1:[rat, monkey, pig, rabbit]
2:[rat, monkey, pig, rabbit, horse]
3:true
4:-1
5:false
subList:[monkey, pig, rabbit]
6:false
7:[rat, monkey, pig, rabbit, horse]
8:rat
9:monkey
```

7.4.2 LinkedList

LinkedList 在内部是采用双向循环链表实现的，插入与删除元素的速度较快，随机访问速度则较慢。LinkedList 单独具有 addFirst()、addLast()、getFirst()、getLast()、removeFirst() 和 removeLast() 方法，这些方法使得 LinkedList 可以作为堆栈、队列和双向队列来使用。这些方法彼此之间只是名称有些差异，或者只存在较少差异，以使得这些名字在特定用法的上下文环境中更加适用（特别是在 Queue 中）。同样，LinkedList 也是线程不安全的。

“栈”通常是后进先出的容器，将元素插入到列表的头部。LinkedList 具有能够直接实现栈功能的方法，因此可以直接将 LinkedList 作为栈来使用。

【例 7.7】 用 LinkedList 实现栈的功能。

MyStack.java

```
package org.container;
import java.util.*;
public class MyStack {
    LinkedList linkedlist = new LinkedList();
    public void push(Object obj){
        linkedlist.addFirst(obj);
    }
    public Object pop() { // 返回栈中第一个元素，并删除栈中该元素
        return linkedlist.removeFirst();
    }
    public Object peek() { // 返回栈中第一个元素
        return linkedlist.getFirst();
    }
    public boolean empty() { // 判断栈是否为空
        return linkedlist.isEmpty();
    }
    public String toString() {return linkedlist.toString();}
    public static void main(String[] args){
        MyStack ms=new MyStack();
        ms.push("Spring");    ms.push("Summer");
        ms.push("Autumn");    ms.push("Winter");
        System.out.println(ms.pop());
        System.out.println(ms.peek());
        System.out.println(ms.pop());
        System.out.println(ms.empty());
        System.out.println(ms);
    }
}
```

程序运行结果：

```
Winter
Autumn
Autumn
false
[Summer, Spring]
```

7.5 Map (映射)

Map (映射) 是一种把键对象和值对象进行映射的集合, 它的每一个元素都包含一对键对象和值对象, 而值对象仍可以是 Map 类型, 依次类推, 这样就形成了多级映射。向 Map 容器中加入元素时, 必须提供一对键对象和值对象。从 Map 容器中检索元素时, 只要给出键对象, 就会返回对应的值对象。在 Map 中, 每个键最多只能映射一个值。

7.5.1 HashMap

HashMap 是基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作, 并允许使用 null 值和 null 键。此类不保证映射的顺序。此实现假定哈希函数将元素适当地分布在各桶之间, 可为基本操作 (get 和 put) 提供稳定的性能。迭代 collection 视图所需的时间与 HashMap 实例的“容量”(桶的数量) 及其大小 (键-值映射关系数) 之和成比例。所以, 如果迭代性能很重要, 则不要将初始容量设置得太高 (或将加载因子设置得太低)。

HashMap 的实例有两个参数影响其性能: 初始容量和加载因子。容量是哈希表中桶的数量, 初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时, 则要对该哈希表进行 rehash 操作 (重建内部数据结构), 从而哈希表将具有大约两倍的桶数。通常, 默认加载因子 (0.75) 在时间和空间成本上寻求一种折中。加载因子过高虽然减少了空间开销, 但同时也增加了查询成本。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子, 以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子, 则不会发生 rehash 操作。如果很多映射关系要存储在 HashMap 实例中, 则相对于按需执行自动 rehash 操作以增大表的容量来说, 使用足够大的初始容量创建它将使得映射关系能更有效地存储。

HashMap 是基于 hashCode 的, 若想正确使用 HashMap, 则需重写 hashCode() 和 equals() 方法。HashMap 不是线程安全的, 若要线程安全, 可用:

```
Map m = Collections.synchronizedMap(new HashMap());
```

【例 7.8】 设计一个 Java 程序, 统计任意给定的一个字符串中, 每一个英文字母的使用频度。

AlphaDegree.java

```
package org.container;
import java.util.*;
public class AlphaDegree {
    public static void main(String[] args){
        String s = "afasdfassgdfgdfgdsdfg";
        char[] num = s.toCharArray();           // 将字符串转换为 char 数组
        int i = num.length-1;
        HashMap map = new HashMap();           // 创建一个 HashMap 对象
        map.put(num[0], 1);
        for (int k=1; k<=i;k++){
            if(map.containsKey(num[k])){        // 如果在容器中已存在该字母, 字母数加 1
                Integer j = (Integer) map.get(num[k]);
                map.put(num[k], ++j);
            }
        }
    }
}
```

```

        else map.put(num[k], 1);           // 如果不存在，将该字母加入到容器中
    }
    System.out.println(map);
}
}

```

程序运行结果：

```
{f=6, g=5, d=5, s=4, a=3}
```

7.5.2 TreeMap

使用 `SortedMap` 接口，可以确保键处于排序状态。这些功能的实现由 `SortedMap` 接口的方法提供。

- `Comparator comparator()`: 返回当前 `Map` 使用的 `Comparator`；或者返回 `null`，表示以自然方式排序。

- `firstKey()`: 返回 `Map` 中的第一个键。

- `lastKey()`: 返回 `Map` 中的最后一个键。

- `SortedMap subMap(fromKey,toKey)`: 生成此 `Map` 的子集，由键小于 `toKey` 的所有键值对组成。

- `SortedMap tailMap(fromKey)`: 生成此 `Map` 的子集，由键大于或等于 `toKey` 的所有键值对组成。

`TreeMap` 类是接口 `SortedMap` 的唯一实现，并且是基于红黑树的。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 `Comparator` 进行排序，具体取决于使用的构造方法。`TreeMap` 是唯一带有 `subMap()` 方法的 `Map`，它可以返回一个子树。

【例 7.9】 设计一个 Java 程序，所要完成的功能是：对一个由数字和非数字组成的字符串，将其中连续的数字字符转换成一个整数。若连续的数字字符个数超过 4 个，则以 4 个数字字符为一组进行转换。转换生成的整数依次存放在整数数组中。如对字符串“c123yz45!786*+56abc123456789”，分析后的整数数组内容为：123、45、786、56、1234、5678、9。

Statistic.java

```

package org.container;
import java.util.*;
public class Statistic {
    public static void main(String[] args) {
        String s = "c789yz45!786*+56abc123456789";
        TreeMap map = new TreeMap();
        char[] arr = s.toCharArray();           // 将字符串转换为字符数组
        int i = arr.length;                     // 获取字符数组的长度
        int k = 0;
        int p = 0;                              // 记下数字字符的开始位置
        int q = 0;                              // 记下数字字符的结束位置
        boolean b = false;
        for (int j = 0; j < i; j++) {
            k = arr[j];

```



```

        if (47 < k && k < 58) {
            if (b == false) {
                p = j;                // 是数字字符的开始位置，记下该位置
                b = true;
            }
            q = j;
            System.out.print(arr[j]);
            if (q != p) {
                map.put(p, q);        // 将开始位置和结束位置放入到 Map 容器中
            }
        }else{
            b = false;
        }
    }
    System.out.println(map);
    Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator(); // 返回一个迭代器
    int []array= new int[10];
    int a = 0;
    while(it.hasNext()){
        Map.Entry entry = it.next();
        int u = (Integer) entry.getKey();    // 获取键对象
        int v = (Integer)entry.getValue()+1; // 获取值对象
        if((v-u)<=3){
            String s1 = s.substring(u,v);
            array[a++]=Integer.parseInt(s1);
        }else{
            String s2 = s.substring(u,v);
            int g = s2.length();
            int h =0;
            while(h+4<g){
                String s3 = s2.substring(h, h+4);
                array[a++] = Integer.parseInt(s3);
                h= h+4;
            }
            String s4 = s2.substring(h, g);        // 取出最后不足 4 个的数字字符
            array[a++] = Integer.parseInt(s4);
        }
    }
    for(int c = 0;c<a;c++){
        System.out.println(array[c]);
    }
}
}

```

程序运行结果：

```
7894578656123456789{1=3, 6=7, 9=11, 14=15, 19=27}
789
45
786
56
1234
5678
9
```

7.6 泛型

Java SE5 引入了“泛型”的概念。泛型实现了参数化类型的概念，使代码可以应用于多种类型。泛型这个术语的意思是“适用于许多种的类型”，目的是希望类或方法能够具备最广泛的表达能力。

【例 7.10】 指定其持有 Object 类型的对象。

BasicType.java

```
package org.generics;
public class BasicType {
    private Object obj;
    public BasicType(Object obj)
    {    this.obj= obj;    }
    public void setObj(Object obj)
    {    this.obj = obj;    }
    public Object getObj()
    {    return obj;    }
    public static void main(String[] args) {
        BasicType type = new BasicType(new A());
        A a = (A)type.getObj();                // 类型转换
        type.setObj(new Double(3.14));
        Double d  = (Double)type.getObj();
        type.setObj(new String("before use generics"));
        String s = (String)type.getObj();
        Double b= (Double)type.getObj();        // 运行时异常
    }
}
class A {
    A(){
        System.out.println("A");
    }
}
```

程序运行结果：

```
AException in thread "main" java.lang.ClassCastException:
    at org.generics.BasicType.main(BasicType.java:20)
```

说明：BasicType 类可以存储任何类型的对象，但要想得到存储的对象，必须强制转换到

正确的类型。例如，当试图把存储的 `String` 对象强制转换为 `Double` 类型时，编译器不会检查到错误，但在运行时会抛出 `ClassCastException` 异常。编译期能检查出的错误就不应该等到运行时才被发现。Java 泛型机制正好可以解决此类问题，它要求编译期的严格类型检查。

【例 7.11】 使用泛型类指定其持有的对象的类型。

BasicGeneric.java

```
package org.generics;

public class BasicGeneric<T> {                                // T 是类型参数
    private T a;
    public BasicGeneric(T a)
    {    this.a = a;    }
    public void set(T a)
    {    this.a = a;    }
    public T get()
    {    return a;    }
    public static void main(String[] args){
        BasicGeneric<B> generic1  = new BasicGeneric<B>(new B());
        B b = generic1.get();
        BasicGeneric<Double> generic2  = new BasicGeneric<Double>(3.14);
        Double d = generic2.get();
        System.out.println(d);
        BasicGeneric<String> generic3  = new BasicGeneric<String>("use generic");
        String s = generic3.get();
        System.out.println(s);
    }
}

class B {
    B(){
        System.out.println(" Class B");
    }
}
```

程序运行结果：

```
Class B
3.14
use generic
```

在程序 `BasicGeneric.java` 中，`T` 是类型参数，所以在创建 `BasicGeneric` 对象时，必须指明想持有什么类型的对象，将其置于尖括号内。例如下面的语句：

```
BasicGeneric<B> generic1  = new BasicGeneric<B>(new B());
```

然后，就只能在 `generic1` 中存入该类型的对象了，这时的 `T` 类型就是引用类型 `B`，所以通过 `get()` 方法就能自动地获得正确的类型，而无须强制类型转换。在创建 `generic2` 对象时，`T` 又变成 `Double` 类型。

其实，一个泛型类就是具有一个或多个类型变量的类。即泛型类可以带有两个及以上类型参数，参数之间用逗号分隔。

【例 7.12】 带有两个类型参数 `T1`、`T2` 的泛型类。

ComplexGeneric.java

```

package org.generics;

public class ComplexGeneric<T1, T2> {
    private T1 first;
    private T2[] second;           // 数组类型
    public void setFirst(T1 f)
    {    this.first = f;    }
    public T1 getFirst()
    {    return first;    }
    public void setSecond(T2[] s)
    {    this.second = s;    }
    public T2[] getSecond()
    {    return second;    }
    public static void main(String[] args) {
        ComplexGeneric<Integer, String> generic = new ComplexGeneric<Integer, String>();
        String[] season = { "spring", "summer", "autumn", "winter" };
        generic.setFirst(new Integer(9));
        generic.setSecond(season);
        System.out.println(generic.getFirst());
        String[] strs = generic.getSecond();
        for (String str : strs) {
            System.out.println(str);
        }
    }
}

```

程序运行结果：

```

9
spring
summer
autumn
winter

```

7.7 通配符与受限通配符

考虑下面的一个简单的泛型类，在这个泛型类中只有简单的 `setXXX()` 和 `getXXX()` 方法。

```

public class Generics<T> {
    private T obj;
    public void setObj(T obj)
    {    this.obj = obj;    }
    public T getObj()
    {    return obj;    }
}

```

可以给上面的这个泛型类定义两个引用：

```

Generics<Integer> gen1 = null;
Generics<String> gen2 = null;

```

那么 `gen1` 就只接收 `Generics<Integer>` 的对象，而 `gen2` 只接收 `Generics<String>` 的对象。现在有一个需求，希望有一个名称为 `gen` 的引用可以接收所有下面的对象。

```
gen = new Generics<ArrayList>();
gen = new Generics<LinkedList>();
```

简单地说，参数化类型必须是 **List** 类型或其子类型，要满足这种要求，可以使用 “?” 通配符，并使用 “extends” 关键字限定参数化类型。例如下面的语句：

```
Generics<? extends List> gen = null;
gen = new Generics<ArrayList>();
gen = new Generics<LinkedList>();
```

如果指定的不是 **List** 的类型或者子类型，则编译器会报告错误。例如下面的语句：

```
Generics<? extends List> gen = new Generics<HashMap>();
```

如果只指定了 `<?>` 而不使用 “extends” 关键字，则可以是 **Object** 类及其子类，也就是所有的类了。那么为什么不直接使用 **Generics** 呢？何必要用 **Generics<?>**？通过使用通配符，可限制对它加入新的信息，只能获取它的信息或是移除它的信息。例如下面的语句：

```
Generics<String> gen = new Generics<String>();
gen.setObj("cat");
Generics<?> gen2 = gen;
System.out.println(gen2.getObj());           // 可以获取信息
gen2.setObj(null);                           // 可通过 gen2 来移去 gen 的信息
gen2.setObj("dog");                           // 不可通过 gen2 来设定新的信息给 gen
```

所以使用 `<?>` 或是 `<? extends SomeClass>` 的方式，意味着只能通过该名称来取得所引用的对象的信息，或者是移除某些信息，但不能增加它的信息。因为只知道当中放置的是 **SomeClass** 的子类，但不确定是什么类的对象，编译器不让加入新的对象。理由是，如果可以加入新的对象的话，那么就记得记住取回的对象是什么类型，然后转换为原来的类型方可进行操作，这就失去了使用泛型的意义。

如果要限定父类的类型，可以使用 “?” 通配符，并使用 “super” 关键字。例如下面的语句：

```
Generics<? super LinkedHashSet> ge = null;
ge = new Generics<HashSet>();
```

【例 7.13】 通配符与受限通配符的使用。

GenericsTest.java

```
package org.generics;
import java.util.*;
public class GenericsTest<T>{
    private T value;
    public T getValue()
    {    return value;    }
    public void setValue(T value)
    {    this.value = value;}
    public static void main(String[] args) {
        GenericsTest<? extends Map> gen = null;
        gen = new GenericsTest<TreeMap>();
        gen = new GenericsTest<HashMap>();
        //gen = new GenericsTest<ArrayList>();           // 不是 Map 的子类型
        GenericsTest<? super LinkedHashMap> gen2 = null;
        gen2 = new GenericsTest<HashMap>();             // HashMap 是 LinkedHashMap 的父类
        GenericsTest<String> gen3 = new GenericsTest<String>();
    }
}
```

```

        gen3.setValue("java generic test");
        System.out.println(gen3.getValue());
        GenericsTest<? extends Object> gen4 = gen3;
        System.out.println(gen4.getValue());
        gen4.setValue(null);
        System.out.println(gen4.getValue());
        //gen4.setValue("hello");           // 不能加入信息
    }
}

```

程序运行结果：

```

java generic test
java generic test
null

```

7.8 综合实例

【例 7.14】 向容器中查询某一学生信息，如果存在则将该学生信息打印出来，并对容器中的 5 个学生信息依总学分升序打印出来。

思路：由于 `TreeMap` 容器中的值对象是 `Student`，需要一一取出它们，计算总学分，再以总学分作为键对象放入到 `TreeMap` 容器中，值对象依然是 `Student`。`TreeMap` 容器会自动以自然顺序对键值对进行排序。

MapTest.java

```

package test;
import java.util.*;
class Student {
    public int id;           // 学号
    public String name;      // 姓名
    public int math_score;   // 数学成绩
    public int english_score; // 英语成绩
    public int computer_score; // 计算机成绩
    public Student(int id,String name,int math_score,int english_score,int computer_score){
        this.id = id;
        this.name = name;
        this.math_score = math_score;
        this.english_score= english_score;
        this.computer_score = computer_score;
    }
}
public class MapTest {
    public static void main(String[] args) {
        TreeMap<Integer, Student> map = new TreeMap<Integer, Student>();
        int total = 0;
        int[] grade = new int[5];
        Student[] s = new Student[5];
    }
}

```

```

s[0] = new Student(100001, "王军", 85, 75, 95);
s[1] = new Student(100002, "李计", 90, 70, 80);
s[2] = new Student(100003, "严红", 92, 80, 80);
s[3] = new Student(100004, "马莉", 80, 87, 76);
s[4] = new Student(100005, "刘燕", 80, 70, 60);
map.put(100001, s[0]); // 将对象放入到容器中
map.put(100002, s[1]);
map.put(100003, s[2]);
map.put(100004, s[3]);
map.put(100005, s[4]);
int[] arr = new int[5];
int i = 0;
int j = Integer.parseInt(args[0]); // 将字符串转换为整型
System.out.println("学号          姓名          计算机成绩          数学成绩
          英语成绩          总学分");
if(map.containsKey(j)){
    System.out.println("你要查找的学生信息是：");
    Student stu = map.get(j); // 从容器中取出一个学生
    System.out.print(stu.id+"          ");
    System.out.print(stu.name+"          ");
    System.out.print(stu.computer_score+"          ");
    System.out.print(stu.english_score+"          ");
    System.out.print(stu.math_score+"          ");
    total = stu.computer_score + stu.english_score + stu.math_score; // 计算总学分
    System.out.print(total);
}
System.out.println();
TreeMap<Integer, Student> tp = new TreeMap<Integer, Student>();
Iterator<Map.Entry<Integer, Student>> it = map.entrySet().iterator(); // 返回一个迭代器
System.out.println("按总学分排序前：");
while (it.hasNext()) {
    Map.Entry entry = it.next();
    arr[i] = (Integer) entry.getKey(); // 获取键对象
    s[i] = (Student) entry.getValue(); // 获取值对象
    System.out.print(s[i].id+"          ");
    System.out.print(s[i].name+"          ");
    System.out.print(s[i].computer_score+"          ");
    System.out.print(s[i].english_score+"          ");
    System.out.print(s[i].math_score+"          ");
    total = s[i].computer_score + s[i].english_score + s[i].math_score;
    System.out.print(total);
    System.out.println();
    grade[i] = total;
    tp.put(grade[i], s[i]);
    i++;
}
i=0;

```

```
System.out.println("按总学分排序后: ");
Iterator<Map.Entry<Integer, Student>> iter = tp.entrySet().iterator();
while(iter.hasNext()){
    Map.Entry entry1 = iter.next();
    arr[i] = (Integer) entry1.getKey();
    s[i] = (Student) entry1.getValue();
    System.out.print(s[i].id+"      ");
    System.out.print(s[i].name+"      ");
    System.out.print(s[i].computer_score+"      ");
    System.out.print(s[i].english_score+"      ");
    System.out.print(s[i].math_score+"      ");
    total = s[i].computer_score + s[i].english_score + s[i].math_score;
    System.out.print(total);
    System.out.println();
}
}
```

右击“MapTest.java”，选择“Run As”→“Run Configurations”，如图 7.2 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_07”，在“Main class”栏中选择“MapTest”，选择“Arguments”标签页，在“Program arguments”栏中输入“100003”，然后单击“Run”按钮，运行程序。

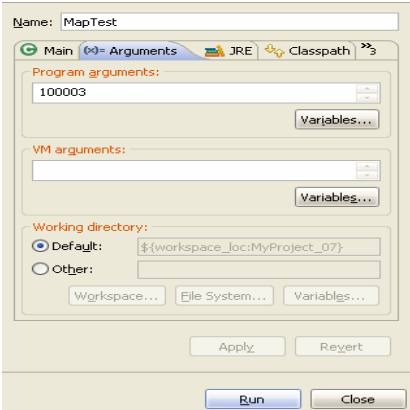


图 7.2 查询学生信息

程序运行结果:

学号	姓名	计算机成绩	数学成绩	英语成绩	总学分
你要查找的学生信息是:					
100003	严红	80	80	92	252
按总学分排序前:					
100001	王军	95	75	85	255
100002	李计	80	70	90	240
100003	严红	80	80	92	252
100004	马莉	76	87	80	243
100005	刘燕	60	70	80	210
按总学分排序后:					
100005	刘燕	60	70	80	210
100002	李计	80	70	90	240
100004	马莉	76	87	80	243
100003	严红	80	80	92	252
100001	王军	95	75	85	255

第 8 章 Java输入/输出系统

输入/输出系统（I/O）处理是程序设计语言中很重要的部分，输入/输出的多样性，会使程序有更广泛的应用。Java 不是在语言层面上对输入/输出提供支持，而是将这个任务交由类库来完成。类库设计者可充分、灵活地设计各种输入/输出技术。

在 Java 中，把一组有序的数据序列称为流，流又分为输入流和输出流两种。可从中读出一系列字节的对象称为输入流，而能向其中写入一系列字节的对象称为输出流。输入流和输出流都是相对的概念，是相对于程序而言的。程序从输入流读取数据，向输出流写入数据。如果数据流中最小的数据单元是字节，那么这种流为字节流；如果数据流中最小的数据单元是字符，那么这种流是字符流。在 I/O 类库中，java.io.InputStream 和 java.io.OutputStream 分别表示字节输入流和字节输出流，java.io.Reader 和 java.io.Writer 分别表示字符输入流和字符输出流。流也可以分为节点流类和过滤流类。用于直接操作目标设备所对应的流叫节点流；通过一个间接流类去调用节点流类，以达到更加灵活方便地读/写各种类型的数据，这个间接流就是过滤流。

8.1 字节流

在 java.io 包中，java.io.InputStream 表示字节输入流，它是抽象类，不能实例化。InputStream 类的作用是用来表示那些从不同数据源产生输入的类。这些数据源有：字节数组、String 对象、文件、管道及其他数据源。每一种数据源都有相应的 InputStream 子类。输入流的类层次结构如图 8.1 所示。

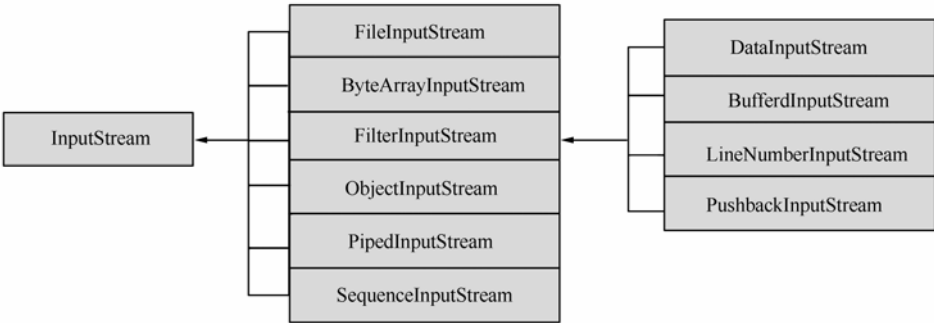


图 8.1 输入流的类层次结构

InputStream 中的读取数据的方法如下。

● abstract int read() throws IOException

功能：读取一个字节数据，并返回读到的数据。如果返回-1，则表示读到了输入流的末尾。

● int read(byte[] b) throws IOException

功能：从输入流中读取一定数量的字节，并将其存储在缓冲区数组 **b** 中，并以整数形式返回实际读取的字节数。如果返回-1，则表示读到了输入流的末尾。

- `int read(byte[] b, int off, int len) throws IOException`

功能：将数据读入一个字节数组，同时返回实际读取字节数。如果返回-1，则表示读到了输入流的末尾。`off` 指定在数组 **b** 中存放数据的起始偏移位置，`len` 指定读取的最大字节数。如果返回-1，则表示读到了输入流的末尾。

- `long skip(long n) throws IOException`

功能：跳过和放弃此输入流中的 *n* 个字节，返回跳过的实际字节数。如果 *n* 为负，则不跳过任何字节。默认实现是此类的 `skip` 方法创建一个 `byte` 数组，然后重复将字节读入其中，直到读够 *n* 个字节或已到达流末尾为止。

- `int available() throws IOException`

功能：返回此输入流下一个方法调用可以不受阻塞地从此输入流读取（或跳过）的估计字节数。

- `void close()`

功能：关闭输入流，释放和这个流相关的系统资源。

`java.io.OutputStream` 表示字节输出流，它也是抽象类，不能被实例化。字节输出流的种类和字节输入流是大致对应的。输出流的类层次结构如图 8.2 所示。

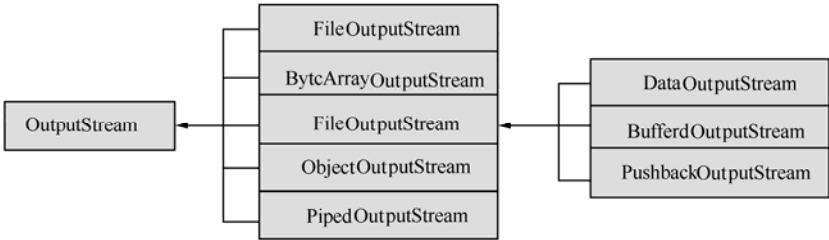


图 8.2 输出流的类层次结构

`OutputStream` 中的写入数据的方法如下。

- `abstract void write(int b) throws IOException`：将 *b* 的最低的一个字节写入此输出流，*b* 的高位字节（3 个）丢弃。
- `void write(byte[] b) throws IOException`：将 *b.length* 个字节从指定的 `byte` 数组写入此输出流。
- `void write(byte[] b, int off, int len) throws IOException`：将指定 `byte` 数组中从偏移量 *off* 开始的 *len* 个字节写入此输出流。
- `void flush() throws IOException`：刷新此输出流并强制写出所有缓冲的输出字节。
- `void close() throws IOException`：关闭此输出流并释放与此流有关的所有系统资源。

8.1.1 文件输入流

`FileInputStream` 类用于从文件读取数据，主要用于二进制文件的读（如读取图像数据之类的原始字节流，要读取文本文件，可考虑使用 `FileReader`），它的构造方法如下。

- `FileInputStream(File file) throws FileNotFoundException`

功能：通过打开一个到实际文件的连接来创建一个 `FileInputStream`，该文件通过文件系统中的 `File` 对象指定。

● `FileInputStream(String name) throws FileNotFoundException`

功能：通过打开一个到实际文件的连接来创建一个 `FileInputStream`，该文件通过文件系统中的路径名 `name` 指定。

【例 8.1】 使用文件输入流把文本文件 `t1.txt` 中的三角形图案输出到屏幕上。

`FileInputStream.java`

```
package org.iostream;
import java.io.*;
class FileInputStreamDemo {
    public static void main(String[] args) {
        String filename;
        int ch = 0;
        filename = "e:/workbench/MyProject_08/src/org/iostream/t1.txt";
        try {
            FileInputStream fis = new FileInputStream(filename);
            while ((ch = fis.read()) != -1) {           // 从文件输入流读取数据
                System.out.print((char) ch);
            }
            fis.close();                                // 关闭文件输入流
        } catch (IOException e) {
            System.out.println("File not found");
        }
    }
}
```

程序运行结果：



8.1.2 文件输出流

`FileOutputStream` 类用于向文件写数据，主要用于二进制文件的写（如用于写入图像数据之类的原始字节流，要写入文本文件，可考虑使用 `FileWriter`），它的构造方法如下。

● `FileOutputStream(String name) throws FileNotFoundException`

功能：创建一个向具有指定名称的文件中写入数据的输出文件流，若文件已存在，则文件中的内容被清除。

● `FileOutputStream(String name, boolean append) throws FileNotFoundException`

功能：创建一个向具有指定 `name` 的文件中写入数据的输出文件流。如果第二个参数为 `true`，则以添加方式写入字节，文件中的原有内容不会被清除。

● `FileOutputStream(File file) throws FileNotFoundException`

功能：创建一个向指定 `File` 对象表示的文件中写入数据的文件输出流。

● `FileOutputStream(File file, boolean append)` throws `FileNotFoundException`

功能：创建一个向指定 `File` 对象表示的文件中写入数据的文件输出流。如果第二个参数为 `true`，则将字节写入文件末尾处，而不是写入文件开始处。

【例 8.2】 使用文件输出流将 100~200 之间能被 3 整除的数写入到文本文件中，要求每 10 个数一行。

FileOutputStreamDemo.java

```
package org.iostream;
import java.io.*;
public class FileOutputStreamDemo {
    public static void main(String[] args) throws IOException {
        int n = 0; int num = 0; int i = 0;
        String filename = "e:/workbench/MyProject_08/src/org/iostream/t2.txt";
        FileOutputStream fos = null;
        FileInputStream fis = null;
        try {
            fos = new FileOutputStream(filename,true);
            for (n =100;n<=200;n++){
                if (n % 3 ==0){
                    i++;
                    String str = String.valueOf(n);           // 返回整型值的字符串表示形式
                    String str1 = str+" ";                    // 两数之间保留一定空隙
                    byte[] buff = str1.getBytes();             // 把字符串转换为字节数组
                    fos.write(buff);
                    if(i%10==0){
                        str = "\r\n";                          // 回车换行
                        byte[] buf = str.getBytes();
                        fos.write(buf);
                    }
                }
            }
            fos.close();
        } catch (FileNotFoundException e1) {
            System.out.println(e1);
        } catch (IOException e2) {
            System.out.println(e2);
        }
    }
}
```

运行程序，100~200 之间能被 3 整除的数写入到 t2.txt 文件中。t2.txt 文件中的内容如下。

102	105	108	111	114	117	120	123	126	129
132	135	138	141	144	147	150	153	156	159
162	165	168	171	174	177	180	183	186	189
192	195	198							

8.2 过滤流

前面介绍的节点流类只能操作字节数据，在实际应用中，可能操作各种类型的数据。如文件流类，就必须先将其他类型的数据转换成字节数组后写入文件，或将从文件中读取的字节转换成其他类型。这时，需要一个中间类，这个中间类提供了读/写各种数据类型的方法。当需要写入其他类型的数据时，只要调用中间类中的对应方法，这个中间类的方法将其他数据类型转换成字节数组，然后调用底层的节点流将这个字节数组写入目标设备。这个中间类称为过滤流类或处理流类。

过滤流类分为过滤流输入流 `FilterInputStream` 类和过滤流输出流 `FilterOutputStream` 类。两大流类又分别有多个子类。

8.2.1 缓冲流类

`BufferedInputStream` 类和 `BufferedOutputStream` 类将一个字节流转变成为一个带缓冲的字节流，从而使流具有缓冲能力并支持 `skip()` 和 `reset()` 操作。当创建 `BufferedInputStream` 类的对象时，在内部同时创建一个字节数组用于缓冲。当从字节输入流用 `read()` 或 `skip()` 操作时，自动从内部缓冲区读取并在需要时自动从最初的字节输入流中一次读取多个字节来填充内部缓冲。由于提供了缓冲机制，把任意的输入流或输出流串接到缓冲流上会获得性能的提高。生成 `BufferedInputStream` 和 `BufferedOutputStream` 对象时，除了要指定所串接的输入、输出字节流外，还可以指定内部缓冲区的大小，默认缓冲区大小是 8 192 字节。可以将缓冲区大小设定为内存页或磁盘块等的整数倍，以提高性能。

对于 `BufferedInputStream`，当读取数据时，数据按块读入缓冲区，其后的读操作则直接访问缓冲区。在使用 `BufferedOutputStream` 进行输出时，数据首先写入缓冲区，当缓冲区满时，其中的数据写入所串接的输出流。用该类所提供的方法 `flush()` 可以强制将缓冲区的内容全部写入输出流。

`BufferedInputStream` 类的构造方法如下。

- `BufferedInputStream(InputStream in)`

功能：创建一个 `BufferedInputStream` 并保存其参数，创建一个内部缓冲区数组并将其存储在 `buf` 中。

- `BufferedInputStream(InputStream in, int size)`

功能：创建具有指定缓冲区大小的 `BufferedInputStream` 并保存其参数，创建一个长度为 `size` 的内部缓冲区数组并将其存储在 `buf` 中。

`BufferedOutputStream` 类的构造方法如下。

- `BufferedOutputStream(OutputStream out)`

功能：创建一个新的缓冲输出流，以将数据写入指定的底层输出流。

- `BufferedOutputStream(OutputStream out, int size)`

功能：创建一个新的缓冲输出流，以将具有指定缓冲区大小的数据写入指定的底层输出流。

【例 8.3】 设计一个 Java 程序，将数 p 之内的所有质数写入文本文件中，要求 s 个数一行。

TestPrime.java

```
package org.iostream;
import java.io.*;
public class TestPrime {
```

```

BufferedInputStream bis =null;
BufferedOutputStream bos = null;
String filename = "e:/workbench/MyProject_08/src/org/iostream/t3.txt";
static int s,p;
boolean isPrime(int n) {
    for (int i = 2; i <= n / 2; i++)
        if (n % i == 0) return false;
    return true;
}
void printPrime(int m) throws IOException {
    bos = new BufferedOutputStream(new FileOutputStream(filename));
    int j = 0;
    for (int i = 2; i <= m; i++){
        if (isPrime(i)){
            j++;
            if(j%s==0){
                String s= String.valueOf(i)+" ";
                bos.write(s.getBytes());           // 将字符串转换为字节数组
                bos.write("\r\n".getBytes());       // 写入回车换行符
            }else {
                String s= String.valueOf(i)+" ";
                bos.write(s.getBytes());
            }
        }
    }
    bos.flush();           // 刷新输出流
    bos.close();           // 关闭输出流
}
void getPrime() throws Exception{
    bis = new BufferedInputStream(new FileInputStream(filename));
    int c =bis.read();      // 读取输入流
    while( c!= -1){
        char ch = (char)c;  // 将整型转换为 char 类型
        System.out.print(ch);
        c = bis.read();
    }
    bis.close();
}
public static void main(String[] args) throws Exception {
    TestPrime pn = new TestPrime();
    p = Integer.parseInt(args[0]);           // 将字符串类型转换为整型
    s = Integer.parseInt(args[1]);
    pn.printPrime(p);           // 打印出 100 之内的所有质数
    pn.getPrime();              // 读取文本文件中的 100 个质数
}
}

```

右击“TestPrime.java”，选择“Run As”→“Run Configurations”，如图 8.3 所示，选择 Main 标签页，在“Project”栏中选择“MyProject_08”，在“Main class”栏中选择“TestPrime”，选择“Arguments”标签页，在“Program arguments”栏中输入“100 10”，然后单击“Run”按钮，运行程序。100 之内所有质数将被写入到文本文件 t3.txt 中。再通过输入流将刚才写入的质数打印到控制台上。

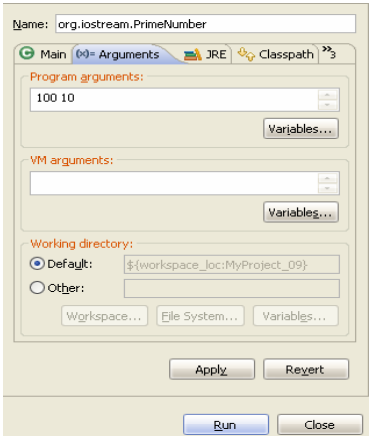


图 8.3 打印 100 之内所有质数

文本文件 t3.txt 中的内容如下。

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97
```

说明：在关闭过滤流时，会自动关闭所串接的底层字节流。

8.2.2 数据流类

DataInputStream 类和 DataOutputStream 类提供了读/写各种基本数据类型以及 String 对象的方法。DataInputStream 类的所有读方法都以“read”开头，例如下面的方法：

- readByte(): 从输入流中读取 1 个字节，把它转换为 byte 类型的数据。
- readFloat(): 从输入流中读取 4 个字节，把它转换为 float 类型的数据。
- readLong(): 从输入流中读取 8 个字节，把它转换为 long 类型的数据。
- readUTF(): 从输入流中读取若干个字节，把它转换为采用 UTF-8 字符编码的字符串。

注意：readUTF()方法能够从输入流中读取采用 UTF-8 字符编码的字符串。UTF-8 字符编码是 Unicode 字符编码的变体。Unicode 字符编码把所有字符都存储为两个字节。如果实际上要存储的字符都是 ASCII 字符（只占 7 位），则采用 Unicode 字符编码会浪费存储空间。UTF-8 字符编码能够更加有效地利用存储空间，它对 ASCII 字符采用一个字符形式的编码，对非 ASCII 字符则采用两个或两个以上字符形式的编码。

DataOutputStream 类的所有方法都以“write”开头，例如下面的方法：

- writeByte(): 向输出流中写入 byte 类型的数据。
- writeLong(): 向输出流中写入 long 类型的数据。
- writeFloat(): 向输出流中写入 float 类型的数据。

- writeUTF(): 向输出流中写入按 UTF 编码的数据。
- 【例 8.4】 用 DataInputStream 读取 DataOutputStream 写入的数据，保证正确读取格式化数据。

DataStreamDemo.java

```
package org.iostream;
import java.io.*;

public class DataStreamDemo {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream(
            "e:/workbench/MyProject_08/src/org/iostream/t4.txt");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        DataOutputStream dos = new DataOutputStream(bos);
        dos.writeByte(75);    dos.writeLong(10000);
        dos.writeChar('a');   dos.writeUTF("北京");
        dos.close();

        FileInputStream fis = new FileInputStream(
            "e:/workbench/MyProject_08/src/org/iostream/t4.txt");
        BufferedInputStream bis = new BufferedInputStream(fis);
        DataInputStream dis = new DataInputStream(bis);
        System.out.print(dis.readByte() + " ");
        System.out.print(dis.readLong() + " ");
        System.out.print(dis.readChar() + " ");
        System.out.print(dis.readUTF() + " ");
        dis.close();
    }
}
```

程序运行结果：

75 10000 a 北京

该程序的数据流向如图 8.4 所示。

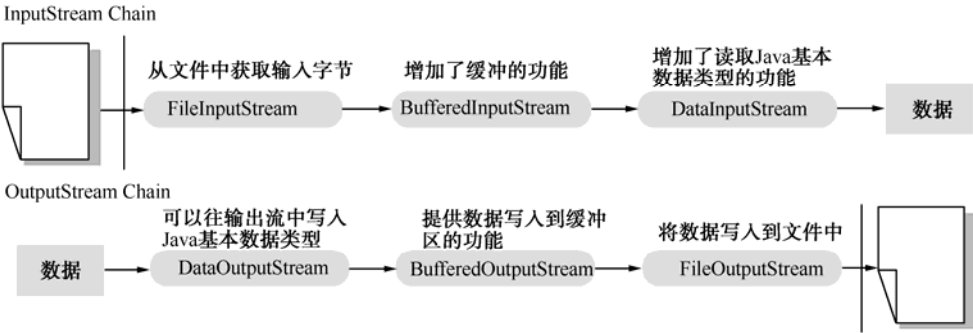


图 8.4 数据流向

8.2.3 PrintStream类

PrintStream 类为其他输出流添加了功能，使它们能够方便地显示各种数据值表示形式。PrintStream 类支持自动刷新功能，这意味着可在写入 byte 数组之后自动调用 flush() 方法，可调用其中一个 println()方法，或写入一个换行符或字节("\n")。PrintStream 类显示的所有字符都使用平台的默认字符编码转换为字节。

PrintStream 类的写数据方法都以“print”开头，例如下面的方法：

- print(int i): 向输出流写入一个 int 类型的数据，按照平台默认的字节编码，并将 String.valueOf(int i)全部写入这些字节。
- print(String s): 向输出流写入一个 String 类型的数据，采用本地操作系统的默认字符编码。
- println(int i): 向输出流写入一个 int 类型的数据和换行符。
- println (String s): 向输出流写入一个 String 类型的数据，采用本地操作系统的默认字符编码和换行符。

【例 8.5】 把 PrintStream 流串接到 FileOutputStream 流，向 t5.txt 文件中写入杨辉三角形，要求写入 10 行。

PrintStreamDemo.java

```
package org.iostream;
import java.io.*;
public class PrintStreamDemo {
    public static void main(String[] args) {
        int[][] a = new int[10][10];
        PrintStream ps = null;
        try {
            FileOutputStream fos = new FileOutputStream(
                "e:/workbench/MyProject_08/src/org/iostream/test3.txt");
            ps = new PrintStream(fos);
            if (ps != null) {
                System.setOut(ps); // 使标准输出重定向
            }
            int i = 0;    int j = 0;
            for (i = 0; i < 10; i++) {
                a[i][i] = 1; // 使对角线元素为 1
                a[i][0] = 1; // 使第一列元素为 1
            }
            for (i = 2; i < 10; i++)
                for (j = 1; j <= i - 1; j++){
                    a[i][j] = a[i - 1][j - 1] + a[i - 1][j]; // 上一行中同列和前一列两个数之和
                }
            for (i = 0; i < 10; i++) {
                for (j = 0; j <= i; j++) {
                    System.out.print(a[i][j] + " ");
                }
                System.out.println();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

运行上面的程序，t5.txt 文件中将被写入 10 行杨辉三角形。t5.txt 文件中的内容如下。

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1

```

8.3 字符流

`InputStream` 和 `OutputStream` 类处理的是字节流，数据流中最小单元为一个字节，但在读/写字符文本时就不太方便。Java 语言采用 Unicode 字符编码，对于每一个字符，Java 虚拟机会为其分配两个字节的内存。为了便于读/写采用各种字符编码的字符，`java.io` 包中提供了 `Reader/Writer` 类，它们分别表示字符输入流和字符输出流。

在读/写文本文件时，最主要的问题是进行字符编码的转换。在文本文件中，字符有可能采用各种类型的编码，如 GBK 和 UTF-8 字符编码等。

`String` 类的 `getBytes(String encode)` 方法返回字符串的特定类型的编码，`encode` 参数指定编码类型。`String` 类的不带参数的 `getBytes()` 方法则使用本地操作系统的默认字符编码。

在 Java 程序中，以下两种方式都能获得本地平台的字符编码类型。

- `System.getProperty("file.encoding");` // 在中文操作系统中显示 GBK
- `Charset cs =Charset.defaultCharset();`

`System.out.println(cs);` // 在中文操作系统中显示 GBK

如果操作系统为中文 Windows 操作系统，以上代码一般会显示“GBK”，在中文 Linux 平台上，通常会显示“UTF-8”。`Charset` 类位于 `java.nio.charset` 包中。

`Reader` 类能够将输出流中采用其他编码类型的字节流转换为 Unicode 字符，然后在内存中为这些 Unicode 字符分配内存。`Writer` 类能够把内存中的 Unicode 字符转换为其他编码类型的字节流，再写到输出流中。在默认的情况下，`Reader` 和 `Writer` 会在本地平台的字符编码和 Unicode 字符编码之间进行编码转换。

如果要输入或输出采用特定类型编码的字节流，可以使用 `InputStreamReader` 类和 `OutputStreamWriter` 类。在它们的构造方法中，可以指定输入流或输出流的字符编码。由于 `Reader` 和 `Writer` 采用了字符编码转换技术，Java I/O 系统能够正确地访问采用各种字符编码的文本文件。另外，在为字符分配内存时，Java 虚拟机对字符统一采用 Unicode 字符编码，

因此 Java 程序处理字符具有平台无关性。

8.3.1 FileReader和FileWriter

FileReader 类用于字符文件的读，每次读取一个字符或一个字符数组。**FileWriter** 类用于字符文件的写，每次写入一个字符、一个数组或一个字符串。通常将 **FileReader** 类的对象看成一个以字符为基本单位的无格式的字符输入流，将 **FileWriter** 的对象看成一个以字符为基本单位的无格式的字符输出流。**FileReader** 和 **FileWriter** 只能按照平台默认的字符编码进行字符的读/写，若要指定字符的编码，请使用 **InputStreamReader** 和 **OutputStreamWriter**。

FileReader 的构造方法如下。

- **FileReader(String fileName) throws FileNotFoundException**

功能：在给定从中读取数据的文件名的情况下创建一个新 **FileReader**。

- **FileReader(File file) throws FileNotFoundException**

功能：在给定从中读取数据的 **File** 的情况下创建一个新 **FileReader**。

FileWriter 的构造方法如下。

- **FileWriter(File file,boolean append) throws IOException**

功能：根据给定的 **File** 对象构造一个 **FileWriter** 对象。如果第二个参数为 **true**，则将字符以添加方式写入文件末尾处。若为 **false**，则原有文件内容被清除，以便写入新文件。

- **FileWriter(String fileName,boolean append) throws IOException**

功能：根据给定的文件名以及指示是否附加写入数据的 **boolean** 值来构造 **FileWriter** 对象。若 **append** 为 **false**，则原有文件内容被清除。

【例 8.6】 将九九乘法表写入到文本文件 **t7.txt** 中。

MultiplicationTable.java

```
package org.iostream;
import java.io.*;
public class MultiplicationTable {
    public static void main(String[] args) throws IOException {
        String filename = "e:/workbench/MyProject_08/src/org/iostream/t7.txt";
        FileReader fr = new FileReader(filename);
        FileWriter fw = new FileWriter(filename,true);
        for (int i = 1; i <= 9; i++) {
            for (int j = 1; j <= i; j++) {
                String s = i + "*" + j + "=" + i * j + " ";
                fw.write(s);
            }
            fw.write("\r\n");           // 写入回车换行符
        }
        fw.flush();                   // 刷新输出流
        int c;
        while ((c = fr.read()) != -1){           // 将九九乘法表读取出来
            System.out.print((char)c);
        }
    }
}
```

程序运行结果:

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

8.3.2 **BufferedReader**和**BufferedWriter**

文本行以回车、换行结束字符序列，有时以文本行为基本单位进行文本读取与处理更方便。**BufferedReader** 和 **BufferedWriter** 这个带缓冲的字符流，可用于以文本行为基本单位进行处理的场合。如要求从键盘读入一个整数值 123，可将键盘变成一个带缓冲的字符输入流，一次读入串"123"，然后用 `Integer.parseInt("123")`转换成整数 123。表 8.1 列出了 **BufferedReader** 类的常用方法，其他方法都是重写了 **Reader** 类中所提供的方法。表 8.2 列出了 **BufferedWriter** 类的常用方法，其他方法都是重写了 **Writer** 类中所提供的方法。

表 8.1 **BufferedReader** 类的常用方法

方 法	功 能
<code>BufferedReader(Reader in)</code>	将字符输入流 in 转换成带缓冲的字符流，字符缓冲区大小为系统默认值
<code>BufferedReader(Reader in, int sz)</code>	将字符输入流 in 转换成带缓冲的字符流，字符缓冲区大小为 sz
<code>String readLine() throws IOException</code>	从输入流中读取一行字符，行结束标志为回车符('\r')、换行符('\n')或者连续的回车换行符('\r\n')。若读到流结束，则返回 null。若流中暂时无数据可读，则该方法进入阻塞状态。注意：返回的字符串中不含行结束符

表 8.2 **BufferedWriter** 类的常用方法

方 法	功 能
<code>BufferedWriter(Writer out)</code>	将字符输出流 out 转换成带缓冲的字符输出流，字符缓冲区大小为系统默认值
<code>BufferedWriter(Writer out, int sz)</code>	构造函数：将字符输出流 out 转换成带缓冲的字符输出流，字符缓冲区大小为 sz
<code>void newLine() throws IOException</code>	写入行结束标记，该标记不是简单的换行符('\n')，而是由系统定义的属性 <code>line.separator</code>

【例 8.7】 设计一个 Java 程序，将 100 之内的所有质数写入文本文件中，要求 10 个数一行。

PrimeNumber.java

```
package org.iostream;
```

```

import java.io.*;
public class PrimeNumber {
    BufferedWriter bw =null;
    String filename = "e:/workbench/MyProject_08/src/org/iostream/t8.txt";
    boolean isPrime(int n) {                                // 判断是否是质数
        for (int i = 2; i <= n / 2; i++)
            if (n % i == 0)
                return false;
        return true;
    }
    void printPrime(int m) throws IOException {
        bw = new BufferedWriter(new FileWriter(filename));
        int j = 0;
        for (int i = 2; i <= m; i++){
            if (isPrime(i)){
                j++;
                String s= String.valueOf(i);
                String s1 = s + " ";
                bw.write(s1);                                // 写入到文本文件中
                if(j==10){
                    j=0;
                    bw.newLine();                            // 写入一个行分隔符
                }
            }
        }
        bw.flush();                                         // 刷新输出流
        bw.close();
    }
    public static void main(String[] args) throws IOException {
        PrimeNumber pn = new PrimeNumber();
        pn.printPrime(100);                                // 打印出 100 之内的所有质数
    }
}

```

运行程序，文本文件 t8.txt 中的内容如下。

```

2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97

```

8.4 标准I/O

Java 中的 I/O 流，并不是存在于整个程序运行的生命周期中的。对流的操作完毕时，就应该适时地关闭流。但对于某些操作，例如向控制台不时地输出信息，如果每次都要打开流，再关闭流，这样很不方便。为此，java.lang.System 类中提供以下三个静态常量。

- static final InputStream in

功能：“标准”输入流，流已打开并准备提供输入数据。通常，此流对应于键盘输入或者由主机环境或用户指定的另一个输入源。

- **static final PrintStream out**

功能：“标准”输出流，此流已打开并准备接受输出数据。通常，此流对应于显示器输出或者由主机环境或用户指定的另一个输出目标。

- **static final PrintStream err**

功能：“标准”错误输出流。此流已打开并准备接受输出数据。通常，此流对应于显示器输出或者由主机环境或用户指定的另一个输出目标。

以上三种流都是 Java 虚拟机在启动应用程序时自动创建的，它们存在于整个程序运行的生命周期中。在程序运行的任何时候都可以通过它们来输入或输出数据。程序的所有输入都可以来自于标准输入，所有输出也都可以发送到标准输出，以及所有的错误信息都可以发送到标准错误输出流。

在默认情况下，标准输入流从键盘读取数据，标准输出流和标准错误输出流向控制台输出数据。Java 的 System 类提供了一些简单的静态方法调用，允许对标准输入、输出和错误 I/O 进行重定向。

- **static void setIn(InputStream in):** 对标准输入流重定向。
- **static void setOut(PrintStream out):** 对标准输出流重定向。
- **static void setErr(PrintStream err):** 对标准错误输出流重定向。

【例 8.8】 测试标准 I/O 的重定向，将标准输入附接到文件上，并将标准输出和标准错误重定向到另一个文件上。

StandardIORedirect.java

```
package org.iostream;
import java.io.*;

public class StandardIORedirect {
    public static void main(String[] args) throws IOException {
        PrintStream console = System.out;
        BufferedInputStream in = new BufferedInputStream(new FileInputStream(
            "e:/workbench/MyProject_08/src/org/iostream/StandardIORedirect.java"));
        PrintStream out = new PrintStream(new BufferedOutputStream(new FileOutputStream(
            "e:/workbench/MyProject_08/src/org/iostream/t10.txt")));
        System.setIn(in); // 对标准输入流重定向
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String s;
        while ((s = br.readLine()) != null) // 从 BufferedReader 类中读取一行数据
            System.out.println(s);
        out.close();
        System.setOut(console);
    }
}
```

运行该程序，源程序的代码将被复制到 t10.txt 中。

8.5 File类

File（文件）类既能代表一个特定文件的名称，又能代表一个目录下的一组文件的名称。如果它指的是一个文件集，就可以对此文件集调用 `list()` 方法，这个方法会返回一个字符串数组。File 类可用来查看文件或目录的信息，还可以创建或删除文件和目录。

File 类的构造方法如下。

- `File(String pathname)`

功能：以 `pathname` 为路径创建 File 对象。

- `File(String parent,String child)`

功能：以 `parent` 为父路径，`child` 为子路径创建 File 对象。

File 类的常用方法如下。

- `boolean canRead()`：判断能否对 File 对象所代表的文件进行读。

- `boolean canWrite()`：判断能否对 File 对象所代表的文件进行写。

- `boolean exists()`：判断该 File 对象所代表的文件或目录是否存在。

- `boolean isDirectory()`：判断该 File 对象是否代表一个目录。

- `boolean isFile()`：判断该 File 对象是否代表一个文件。

- `boolean createNewFile()throws IOException`：如果该 File 对象代表文件，并且该文件不存在，则创建该文件。

- `boolean mkdir()`：在文件系统中创建由该 File 对象表示的目录。

- `boolean mkdirs()`：在文件系统中创建由该 File 对象表示的目录。如果该目录的父目录不存在，则创建该目录的所有的父目录。

【例 8.9】 使用 File 类的常用方法。

UseFile.java

```
package org.iostream;
import java.io.*;
import java.util.Date;
public class UseFile {
    public static void main(String args[])throws Exception{
        File dir1=new File("D:/dir1");
        if(!dir1.exists())
            dir1.mkdir();
        File dir2=new File(dir1,"dir2");
        if(!dir2.exists())
            dir2.mkdirs();
        File dir4=new File(dir1,"dir3/dir4");
        if(!dir4.exists())
            dir4.mkdirs();
        File file=new File(dir2,"test.txt");
        if(!file.exists())
            file.createNewFile();
        listDir(dir1);
        deleteDir(dir1);
    }
}
```

```

    }
    public static void listDir(File dir){
        File[] lists=dir.listFiles();
        //*****显示当前目录下包含的所有子目录和文件的名字*****
        String info="目录:"+dir.getName()+"(";
        for(int i=0;i<lists.length;i++)
            info+=lists[i].getName()+" ";
        info+=")";
        System.out.println(info);
        //*****显示当前目录下包含的所有子目录和文件*****
        for(int i=0;i<lists.length;i++){
            File f=lists[i];
            if(f.isFile())
                System.out.println("文件:"+f.getName()+" canRead:"+f.canRead()
                    +" lastModified:"+new Date(f.lastModified()));
            else
                listDir(f);           // 如果为目录，就递归调用 listDir()方法
        }
    }
    public static void deleteDir(File file){
        //*****如果 file 代表文件，就删除该文件*****
        if(file.isFile()){
            file.delete();
            return;
        }
        //*****如果 file 代表目录，先删除目录下的所有子目录和文件*****
        File[] lists=file.listFiles();
        for(int i=0;i<lists.length;i++)
            deleteDir(lists[i]);
        file.delete();           // 删除当前目录
    }
}

```

说明 :以上 UseFile 类的 main()方法首先创建了一个子文件夹树形结构,接着调用 List(dir1)方法查看 dir1 目录及它包含的子目录和文件的信息,最后调用 deleteDir(dir1)方法删除 dir1 目录及它包含的子目录和文件。

程序运行结果:

```

目录:dir1(dir2 dir3 )
目录:dir2(test.txt )
文件:test.txt canRead:true lastModified:Fri Mar 27 20:02:05 CST 2009
目录:dir3(dir4 )
目录:dir4()

```

8.6 综合实例

【例 8.10】 设计一个 Java 程序，将数 p 和数 s 之间的质数写入文本文件中，要求 t 个

数一行， p 、 s 和 t 从键盘输入。计算出每行质数之和，同样打印到同一个文本文件中。

PrimeDemo.java

```
package org.iostream;
import java.io.*;
import java.util.regex.*;
public class PrimeDemo{
    BufferedWriter bw =null;
    String filename = "e:/workbench/MyProject_08/src/org/iostream/t20.txt";
    static int s,p,t;
    static long num;
    boolean isPrime(int n) {
        for (int i = 2; i <= n / 2; i++)
            if (n % i == 0)
                return false;
        return true;
    }
    void printPrime(int m,int n) throws IOException {
        bw = new BufferedWriter(new FileWriter(filename));
        int j = 0;
        for (int i = m; i <= n; i++){
            if (isPrime(i)){
                j++;
                if(j%t==0){
                    System.out.print(" " + i);
                    System.out.print(" " +num);
                    System.out.println();
                    num=num+i;
                    String s= String.valueOf(i);
                    String s1 = s + " ";
                    String s2 = String.valueOf(num);
                    bw.write(s1);
                    bw.write(s2);
                    bw.newLine();
                    num =0;
                }
            }
            else {
                String s= String.valueOf(i);
                String s3 = s + " ";
                bw.write(s3);
                num = num+i;
                System.out.print(" " + i);
            }
        }
        String s4 = String.valueOf(num); // 最后不足 k 个质数的一行的累加和也写入到文本文件中
        System.out.println(" "+num);
    }
}
```

```

        bw.write(s4);
        bw.flush();           // 刷新输出流
        bw.close();
    }
    public static void main(String[] args) throws IOException {
        PrimeDemo pn = new PrimeDemo();
        int[] arr = new int[10];
        BufferedReader br = new BufferedReader(new InputStreamReader(
            System.in));
        String s1;
        while ((s1 = br.readLine()) != null && s1.length() != 0){
            Pattern p1 = Pattern.compile("\\d{1,3}"); // 编译正则表达式，要求 1~3 个数字
            Matcher m = p1.matcher(s1);             // 对字符串进行匹配
            int i = 0;
            while(m.find()) {                         // 寻找与指定模式匹配的下一个子序列
                int j = 0;
                j = Integer.parseInt(m.group());      // 将字符串类型转换为整型
                arr[i]= j;
                i++;
            }
            p = arr[0];                               // 区间整数的起始位置
            s = arr[1];                               // 区间整数的结束位置
            t = arr[2];                               // 每一行的质数个数
            pn.printPrime(p,s);                       // 打印出 100 之内的所有质数
        }
    }
}

```

运行程序，在控制台中输入“101 200 10”，按下回车键。

程序运行结果：

```

101 200 10
101 103 107 109 113 127 131 137 139 149 1067
151 157 163 167 173 179 181 191 193 197 1555
199 199

```

第 9 章 AWT 组件及应用

在可视化程序设计中，人机对话主要是通过窗口和对话框来实现的，因此，设计和构造图形用户界面是软件开发中的一项重要工作。图形用户界面（Graphics User Interface，GUI）生动、直观、操作简单，是程序设计的发展方向。

Java 语言完全支持图形用户界面，它是通过 AWT（Abstract Window Toolkit，抽象窗口工具包）和 Java 基础类（JFC 或更常用的 Swing）来提供这些 GUI 部件的。其中 java.awt 是最原始的 GUI 工具包，存放在 java.awt 包中。现在有许多功能已被 Swing 取代并得到了很大的增强与提高，但是 AWT 中还是包含了最核心的功能。基本 AWT 库将处理用户界面元素的任务委派给每个目标平台（Windows、Solaris、Macintosh 等），由本地 GUI 工具箱负责用户界面元素的创建和动作。例如，如果使用最初的 AWT 的 Java 窗口放置一个文本框，就会有一个底层的“对等体”文本框，用来实际处理文本输入。从理论上说，结果程序可以运行在任何平台上，但感观的效果却依赖于目标平台。AWT 中的图形元素可以分为两类：基本组件 Component 和容器 Container。

9.1 AWT 容器

容器是用来组织其他容器和基本组件的。一个容器可以容纳多个容器和基本组件，并使它们成为一个整体。容器可以简化图形界面的设计与管理。容器本身也是一个组件，具有组件的所有性质。

9.1.1 Window 和 Frame

Window 是不依赖于其他容器而独立存在的容器。Window 有两个子类：Frame 和 Dialog。Frame 带有标题，而且可以调整大小。Dialog 可以被移动，但是不能改变大小。Frame 有一个构造方法 Frame(String title)，通过它可以创建一个以参数为标题的 Frame 对象。Frame 的 add() 方法向容器中加入其他组件。当 Frame 被创建后，它是不可见的，必须通过以下步骤使 Frame 成为可见的。

（1）调用 setSize(int width,int height) 显式设置 Frame 的大小，或者调用 pack() 自动确定 Frame 的大小。pack() 方法确保 Frame 容器中的组件与窗体相适用的大小。

（2）调用 setVisible(true) 方法使 Frame 成为可见。

【例 9.1】 创建一个 Frame 对象并使用 Frame 类的常用方法。

MyFrame.java

```
package org.awt;
import java.awt.*;
public class MyFrame{
    public static void main(String args[]){
        Frame fr=new Frame();
```

```

        fr.setTitle("This is a Frame");           // 设定窗体标题
        fr.setSize(400,300);                       // 设定窗体的宽度为 400，高度为 300
        fr.setBackground(Color.green);             // 设定窗体的背景色为绿色
        fr.setLocation(300,500);                   // 设定窗体左上角的初始位置为(300,500)
        fr.setResizable(false);                    // 设定窗体为不可调整大小
        fr.setVisible(true);                        // 将窗体设为可见
    }
}

```

程序运行结果如图 9.1 所示。

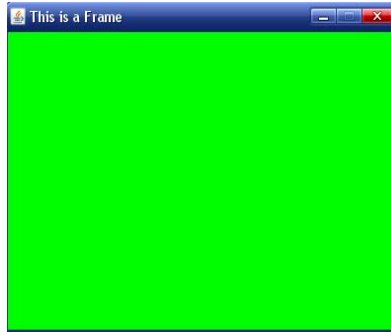


图 9.1 一个 Frame

9.1.2 Panel

面板 **Panel** 是一个通用的容器，它没有边框或其他可见的边界，不能移动、放大、缩小或关闭，不能单独存在，只能存在于其他容器（**Window** 或其子类）中。一个 **Panel** 对象代表一个区域，在这个区域中可以容纳其他的组件。**Panel** 的 **add()**方法向 **Panel** 中添加组件。如要使 **Panel** 成为可见的，必须通过 **Frame** 或 **Window** 的 **add()**方法把 **Panel** 添加到 **Frame** 或 **Window** 中。**Frame** 的 **setBounds (int x,int y,int width,int height)**方法移动组件并调整其大小，由 **x** 和 **y** 指定左上角的新位置，由 **width** 和 **height** 指定新的大小。

【例 9.2】 创建 4 个 **Panel** 对象，并将它们添加到窗体上。

MyMultiPanel.java

```

package org.awt;
import java.awt.*;
public class MyMultiPanel {
    public static void main(String args[]) {
        new NewFrame("This is a panel",300,300,400,300);
    }
}
class NewFrame extends Frame{
    private Panel p1,p2,p3,p4;
    NewFrame(String s,int x,int y,int w,int h){
        super(s);
        setLayout(null);
        p1 = new Panel(null); p2 = new Panel(null);
        p3 = new Panel(null); p4 = new Panel(null);
    }
}

```

```

        p1.setBounds(0,0,w/2,h/2);           // 设置 Panel 对象的大小和位置
        p2.setBounds(0,h/2,w/2,h/2);
        p3.setBounds(w/2,0,w/2,h/2);
        p4.setBounds(w/2,h/2,w/2,h/2);
        p1.setBackground(Color.BLUE);
        p2.setBackground(Color.GREEN);
        p3.setBackground(Color.YELLOW);
        p4.setBackground(Color.MAGENTA);
        add(p1);add(p2);add(p3);add(p4);
        setBounds(x,y,w,h);                 // 设置窗体的大小和位置
        setVisible(true);
    }
}

```

程序运行结果如图 9.2 所示。

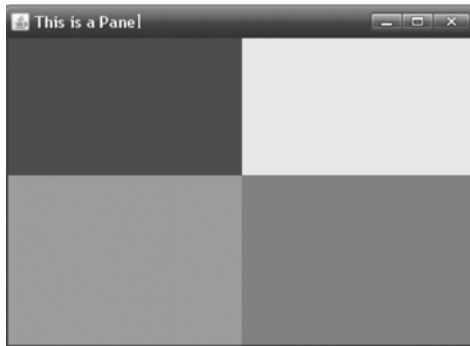


图 9.2 窗体加载 Panel

9.2 布局管理器

Java 为了实现跨平台的特性并获得动态的布局效果，将容器内的所有组件的大小、位置、顺序、间隔等交给布局管理器负责。共有 5 种布局管理器：**FlowLayout**、**BorderLayout**、**GridLayout**、**CardLayout**、**GridBagLayout**。所有的容器都会引用一个布局管理器实例，通过它来自动进行组件的布局管理。

(1) 默认布局管理器

当一个容器被创建后，它有相应的默认布局管理器。**Window**、**Frame** 和 **Dialog** 的默认布局管理器是 **BorderLayout**，**Panel** 的默认布局管理器是 **FlowLayout**。

(2) 取消布局管理器

如果不希望通过布局管理器来管理布局，可以调用容器的 **setLayout(null)** 方法，这样布局管理器就被取消了。但接下来必须调用容器中的每个组件的 **setLocation()**、**setSize()** 或 **setBounds()** 方法，为这些组件在容器中一一定位。

1. 流式布局管理器

FlowLayout 把组件从左向右，从上向下，一个接一个地放到容器中，组件之间的默认间

隔（水平和垂直）为 5 个像素，默认的组件对齐方式为居中。组件的大小由布局管理器根据组件的最佳尺寸来决定。

● **FlowLayout** 的构造方法如下。

```
FlowLayout(int align,int hgap,int vgap)
```

功能：参数 **align** 用来决定组件在每行中相对于容器的边界的对齐方式。可选值有：**FlowLayout.LEFT**（左对齐）、**FlowLayout.RIGHT**（右对齐）和 **FlowLayout.CENTER**（居中对齐）。参数 **hgap** 和参数 **vgap** 分别设定组件之间的水平和垂直间隙。

2. 边界布局管理器

BorderLayout 是 **Window**、**Frame**、**Dialog** 的默认布局管理器，它将容器分成 5 个区域来安排组件：**North**、**South**、**East**、**West**、**Center**。

● **BorderLayout** 的构造方法如下。

```
BorderLayout(int hgap,int vgap)
```

参数 **hgap** 和 **vgap** 分别设定组件之间的水平和垂直间隙。对于采用 **BorderLayout** 的容器，当它用 **add()** 方法添加一个组件时，可以同时指定组件在容器中的区域，如下所示。

```
void add(Component comp,Object constraints)
```

这里的 **constraints** 是 **String** 类型，可选值为 **BorderLayout** 提供的 5 个常量。下面的代码把 **Button** 放在 **Frame** 的东边区域。

```
Frame fr =new Frame();  
fr.add(new Button("bt"),BorderLayout.EAST);
```

如果不指定 **add()** 方法的 **constraints** 参数，在默认的情况下把组件放在中间区域。下面的代码向 **Frame** 的中间区域加入两个 **Button**，但只有最后加入的 **Button** 是可见的。

```
Frame fr =new Frame();  
fr.add(new Button("b1"));  
fr.add(new Button("b2"));  
fr.setSize(200,200);  
fr.setVisible(true);
```

3. 网格布局管理器

GridLayout 将容器分成一个个格子，按行依次排列组件，各组件大小相同。

● **GridLayout** 的构造方法如下。

GridLayout(int rows ,int cols,int hgap,int vgap): **rows** 代表行数，**cols** 代表列数，**hgap** 和 **vgap** 规定网格之间的水平和垂直间隙。

4. 卡片布局管理器

CardLayout 将界面看成一系列卡片，在任何时候只有其中一张卡片是可见的，这张卡片占据容器的整个区域。

● **CardLayout** 的构造方法如下。

```
CardLayout(int hgap,int vgap)
```

参数 **hgap** 表示卡片和容器的左右边界之间的间隙，参数 **vgap** 表示卡片和容器的上下边界的间隙。对于采用 **CardLayout** 的容器，当用 **add()** 方法添加一个组件时，需要同时为组件指定所在卡片的名字，如下所示。

void add(Component comp,Object constraints)

`constraints` 参数是一个字符串，表示卡片的名字。在默认的情况下，容器显示第一个用 `add()` 方法加入容器中的组件，也可以通过 `CardLayout` 的 `show(Container parent,String name)` 方法指定显示哪张卡片，参数 `parent` 指定容器，参数 `name` 指定卡片的名字。

9.3 事件处理机制

当用户与 GUI 交互时，比如单击鼠标，拖动鼠标，敲击键盘，关闭窗口，GUI 应该能够接收到相应的事件并适时处理。每一个可以触发事件的组件都被称做事件源，每一种事件都对应专门的监听器。监听器负责接收和处理这种事件。一个事件源可以触发多种事件，如果它注册了某种事件的监听器，那么这种事件就会被监听器接收和处理。

在 Java 中，监听器对象是一个实现了特定监听器接口的类的实例。事件源是一个能够注册监听器对象并发送事件对象的对象。用户对组件的一个操作，称为一个事件，当事件发生时，事件源将事件对象传递给所有注册的监听器。监听器对象将利用事件对象中的信息决定如何对事件做出响应。在图 9.3 中，`Frame` 是一个事件源，它可以触发键盘事件和鼠标事件等。键盘事件对应一个键盘监听器，它会在键按下和释放时响应。`Frame` 注册了键盘监听器，所以它触发的键盘事件将被处理。对于 `Frame` 触发的鼠标事件，由于没有注册相应的鼠标监听器，所以这种事件不会被处理。

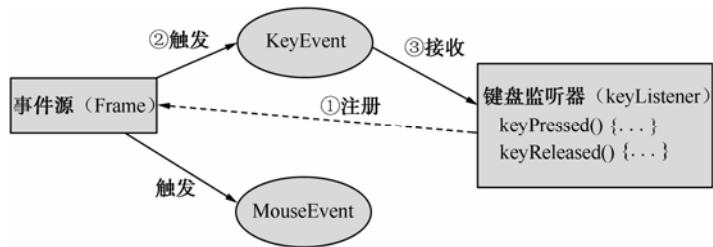


图 9.3 Frame 的触发事件

9.3.1 AWT事件与监听器

每个具体的事件都是某种事件类的实例，事件类包括：`ActionEvent`、`ItemEvent`、`MouseEvent`、`KeyEvent` 和 `WindowEvent` 等。事件类的类层次结构如图 9.4 所示。

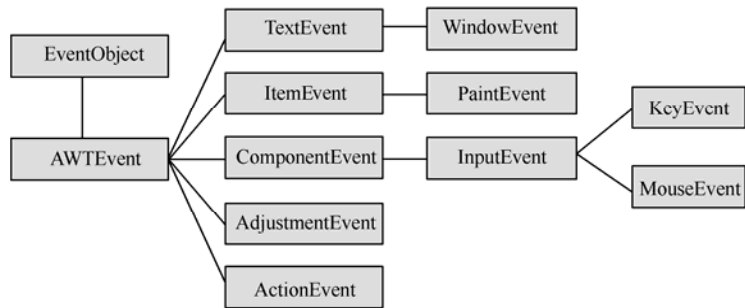


图 9.4 事件类的类层次结构

要处理一个对象所产生的事件，首先必须注册该对象的监听者。java.awt.event 包按照不同的事件类型定义了 11 个监听器接口，每类事件都有对应的事件监听器，监听器是接口，接口中定义了事件发生时可调用的方法，一个类可以实现监听器的一个或多个接口。表 9.1 列出了 AWT 事件的监听器接口。

表 9.1 AWT 事件的监听器接口

事 件 类	说 明	接 口 名	方 法
ActionEvent	动作事件	ActionListener	actionPerformed(ActionEvent e)
ItemEvent	选项事件	ItemListener	itemStateChanged(ItemEvent e)
MouseEvent	鼠标移动事件	MouseMotionListener	mouseGragged(MouseEvent e) mouseMoved(MouseEvent e)
MouseEvent	鼠标事件	MouseListener	mousePressed(MouseEvent e) mouseReleased(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mouseClicked(MouseEvent e)
KeyEvent	键盘事件	KeyListener	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)
FocusEvent	焦点事件	FocusListener	focusGained(FocusEvent e) focusLost(FocusEvent e)
AdjustmentEvent	移动滚动条等 组件	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ContainerEvent	容器事件	ContainerListener	componentAdded(ContainerEvent e) componentRemoved(ContainerEvent e)
ComponentEvent	组件动作事件	ComponentListener	componentMoved(ComponentEvent e) componentHidden(ComponentEvent e) componentResized(ComponentEvent e) componentShown(ComponentEvent e)
TextEvent	文本事件	TextListener	textValueChanged(TextEvent e)
WindowEvent	窗口事件	WindowListener	windowClosing(WindowEvent e) windowOpened(WindowEvent e) windowIconified(WindowEvent e) windowClosed(WindowEvent e) windowDeiconified(WindowEvent e) windowActivated(WindowEvent e) windowDeactivated(WindowEvent e)

9.3.2 窗口事件

WindowEvent 类对应窗口事件，包括用户单击了窗口的“关闭”按钮，窗口得到与失去焦点，窗口最小化等。窗口事件对应的事件监听器是 WindowListener。例如下面的代码：

```
public void windowClosing(WindowEvent e) {           // 关闭窗口
    System.exit(-1);
}
```

9.3.3 内部类实现监听接口

【例 9.3】 Frame 采用 FlowLayout 布局，在 Frame 中加入了 3 个 Button，分别在 3 个 Button 按钮上注册时间监听器。单击“left”按钮，按钮以左对齐方式排列；单击“center”按钮，按钮以居中方式排列；单击“right”按钮，按钮以右对齐方式排列。

InnerListener.java

```
package org.awt;
import java.awt.*;
import java.awt.event.*;
public class InnerListener{
    public static void main(String args[]){
        final Frame f=new Frame("");
        f.setBackground(Color.green);           // 设置窗体的颜色为绿色
        final FlowLayout fl=new FlowLayout();
        f.setLayout(fl);                       // 使 Frame 采用 FlowLayout 布局
        Button leftButton=new Button("left");
        leftButton.setBackground(Color.green);
        leftButton.addActionListener(new ActionListener(){ // 注册事件监听器
            public void actionPerformed(ActionEvent event){
                fl.setAlignment(FlowLayout.LEFT);        // 左对齐
                fl.layoutContainer(f);                    // 使 Frame 重新布局
            }
        });
        Button centerButton=new Button("center");
        centerButton.setBackground(Color.green);
        centerButton.addActionListener(new ActionListener(){ // 注册事件监听器
            public void actionPerformed(ActionEvent event){
                fl.setAlignment(FlowLayout.CENTER);      // 居中对齐
                fl.layoutContainer(f);
            }
        });
        Button rightButton=new Button("right");
        rightButton.addActionListener(new ActionListener(){ // 注册事件监听器
            public void actionPerformed(ActionEvent event){
                fl.setAlignment(FlowLayout.RIGHT);       // 右对齐
                fl.layoutContainer(f);
            }
        });
    }
}
```

```

});
    rightButton.setBackground(Color.green);
    f.add(leftButton);                // 将按钮加入到窗体上
    f.add(centerButton);
    f.add(rightButton);
    f.setSize(300,300);
    f.setVisible(true);
}
}

```

运行程序，单击“left”按钮，按钮以左对齐方式排列；单击“center”按钮，按钮以居中方式排列；单击“right”按钮，按钮以右对齐方式排列。程序的运行结果如图 9.5 所示。



图 9.5 内部类实现监听接口

9.3.4 类自身实现监听接口

同样可以用类的自身实现监听接口。由于 Java 支持一个类实现多个接口，因此类可以实现多个监听接口，类中的组件将类的实例本身注册为监听器。

【例 9.4】 求所有满足条件的四位数，它是 11 的倍数，且第 3 位数加上第 2 位数等于第 4 位数。

ContainerListenerDemo.java

```

package org.awt;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class ContainerListenerDemo extends Frame implements ActionListener {
    // 设置有垂直和水平滚动条的文本区
    static TextArea ta = new TextArea("", 5, 10, TextArea.SCROLLBARS_BOTH);
    public ContainerListenerDemo() {
        Button bt = new Button("求四位数");
        bt.setBackground(Color.cyan);
        bt.addActionListener(this);        // 把 ContainerListener 本身注册为 Button 的监听器
        bt.setBounds(100, 40, 80, 30);
        setLayout(null);                    // 取消布局管理器
        setBackground(Color.cyan);
        setBounds(20, 20, 300, 300);
        ta.setBounds(45, 75, 220, 200);
    }
}

```

```

        add(bt);
        add(ta);
        setVisible(true); // 设置窗口可见
    }

    public void actionPerformed(ActionEvent e) {
        int a = 0; // 千位上的数字
        int b = 0; // 百位上的数字
        int c = 0; // 十位上的数字
        int num = 0; // 这种四位数的个数
        int i = 110;
        for (i = 1000; i < 10000; i++) {
            a = i / 1000;
            b = (i % 1000 - i % 100) / 100;
            c = (i % 100 - i % 10) / 10;
            if ((i % 11 == 0) && (a == b + c)) {
                String str = String.valueOf(i); // 将整型转换为字符串类型
                ta.append(str + " ");
                if (++num % 4 == 0) {
                    ta.append("\n");
                }
            }
        }
        String str2 = String.valueOf(num);
        ta.append(str2 + " ");
    }

    public static void main(String[] args) {
        ContainerListenerDemo cl = new ContainerListenerDemo();
    }
}

```

运行程序，单击“求四位数”按钮，则符合要求的四位数显示在文本区中。程序运行结果如图 9.6 所示。

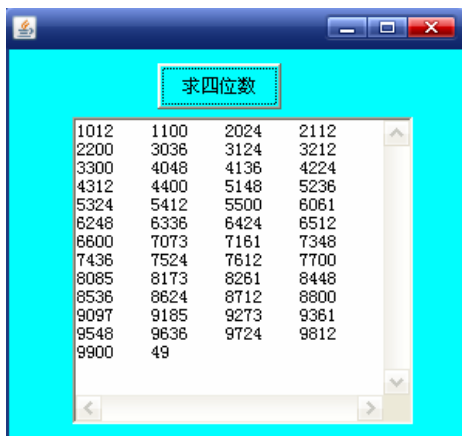


图 9.6 求四位数

9.3.5 外部类实现监听接口

用外部类来实现监听接口，优点是可以使处理事件的代码与创建 GUI 界面的代码分离；缺点是在监听类中无法直接访问组件。在监听类的事件处理方法中不能直接访问事件源，必须通过事件类的 `getSource()` 方法来获得事件源。

【例 9.5】 在窗体上创建 3 个文本框，两个文本框用于运算对象，另一个用于存放计算结果，下拉列表框存放四则运算符号。

MultiplyOperation.java

```
package org.awt;
import java.awt.*;
import java.awt.event.*;
public class MultiplyOperation extends Frame {
    TextField num1;
    TextField num2;
    TextField sum;
    static Choice ch = new Choice(); // 创建下拉列表框
    public static void main(String[] args) {
        MultiplyOperation test = new MultiplyOperation();
        test.operation();
        test.setBackground(Color.cyan); // 设置窗体的颜色
        test.setSize(280, 150); // 设置窗体的位置
        test.addWindowListener(new WindowHandler3()); // 注册事件监听器
    }
    public void operation() {
        num1 = new TextField();
        num2 = new TextField();
        sum = new TextField();
        ch.add("+");
        ch.add("-");
        ch.add("*");
        ch.add("/");
        num1.setColumns(5); // 设置此文本框的列数
        num2.setColumns(5);
        sum.setColumns(5);
        setLayout(new FlowLayout()); // 采用流式布局管理器
        Button btnEqual = new Button("=");
        btnEqual.setBackground(Color.cyan);
        btnEqual.addActionListener(new MyListener1(this)); // 注册事件监听器
        ch.addItemListener(new ChoiceHandler()); // 注册事件监听器
        add(num1); // 将文本框加入到窗体上
        add(ch);
        add(num2);
        add(btnEqual);
        add(sum);
        setVisible(true); // 设置窗体可见
    }
}
```

```

    }
}

class MyListener1 implements ActionListener {
    private MultiplyOperation mulp;
    public MyListener1(MultiplyOperation mulp) {
        this.mulp = mulp;
    }
    public void actionPerformed(ActionEvent e) {
        String s1 = mulp.num1.getText();           // 获取文本框中的内容
        String s2 = mulp.num2.getText();
        int i1 = Integer.parseInt(s1);             // 将字符串类型转换为整型
        int i2 = Integer.parseInt(s2);
        String itm;
        itm = mulp.ch.getSelectedItemAt();
        if (itm.equals("+")) {
            mulp.sum.setText(String.valueOf(i1 + i2));
        } else if (itm.equals("-")) {
            mulp.sum.setText(String.valueOf(i1 - i2));
        } else if (itm.equals("*")) {
            mulp.sum.setText(String.valueOf(i1 * i2));
        } else if (itm.equals("/")) {
            mulp.sum.setText(String.valueOf(i1 / i2));
        }
    }
}

class ChoiceHandler implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        String itm;
        itm = MultiplyOperation.ch.getSelectedItemAt();           // 获取所选项的名称
    }
}

class WindowHandler3 extends WindowAdapter {           // 窗口适配器
    public void windowClosing(WindowEvent e) {           // 关闭窗口
        System.exit(-1);
    }
}

```

运行程序，在文本框中输入 5 和 6，单击“=”按钮，两数相乘的积存放在第三个文本框中，程序的运行结果如图 9.7 所示。



图 9.7 计算两数相乘的积

9.3.6 采用事件适配器

由于监听器实际上就是实现了相应接口的类，而接口一般要求实现许多方法，如接口 `WindowListener` 中有 7 种方法，并不是所有方法都是需要的，若实现该接口，则要求实现其所有的抽象方法。为了简化，Java 语言为一些接口提供了事件适配器（Adapter）。事件适配器是抽象类。通过继承事件适配器，重写需要的方法，不用的方法无须重写。

Java `.awt.event` 包中提供了以下几个事件适配器。

<code>ComponentAdapter</code>	组件适配器
<code>ContainerAdapter</code>	容器适配器
<code>FocusAdapter</code>	焦点适配器
<code>KeyAdapter</code>	键盘适配器
<code>MouseAdapter</code>	鼠标适配器
<code>MouseMotionAdapter</code>	鼠标移动适配器
<code>WindowAdapter</code>	窗口适配器

【例 9.6】 请编写一个程序，在窗口上显示 3 个按钮，按钮上的标题分别是：红色、绿色、蓝色。当按下标题为“红色”的按钮时，则 3 个按钮的标题都变成红色；而按下标题为“绿色”的按钮时，则全部变成绿色；按下标题为“蓝色”的按钮时，则全部变成蓝色。

EventAdapter.java

```
package org.awt;
import java.awt.*;
import java.awt.event.*;
public class EventAdapter{
    public static Button bt1 = new Button("红色");
    public static Button bt2 = new Button("绿色");
    public static Button bt3 = new Button("蓝色");
    public static void main(String[] args) {
        Frame f = new Frame();
        f.setBackground(Color.cyan);
        f.setSize(new Dimension(330, 250));
        f.setLayout(null);
        bt1.setBackground(Color.red);
        bt1.setBounds(new Rectangle(45, 180, 70, 25));
        bt3.setBackground(Color.blue);
        bt2.setBounds(new Rectangle(135, 180, 70, 25));
        bt2.setBackground(Color.green);
        bt3.setBounds(new Rectangle(220, 180, 70, 25));
        ActionListener insert = new InsertAction();
        f.setLocation(300,300);
        f.add(bt2);
        f.add(bt1);
        f.add(bt3);
        f.addWindowListener(new WindowHandler1());
        f.setVisible(true);
        bt1.addActionListener(insert);
        bt2.addActionListener(insert);
```

```

        bt3.addActionListener(insert);
    }
}
//*****方法 windowClosing 就是当窗口关闭时的处理动作*****
class WindowHandler1 extends WindowAdapter {           // 窗口适配器
    public void windowClosing(WindowEvent e) {
        System.exit(1);                                // 关闭窗口
    }
}
class InsertAction implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        String input = event.getActionCommand();        // 返回与此动作相关的命令字符串
        if(input.equals("红色")){
            EventAdapter.bt1.setBackground(Color.red);
            EventAdapter.bt2.setBackground(Color.red);
            EventAdapter.bt3.setBackground(Color.red);
        }
        else if(input.equals("绿色")) {
            EventAdapter.bt1.setBackground(Color.green);
            EventAdapter.bt2.setBackground(Color.green);
            EventAdapter.bt3.setBackground(Color.green);
        }
        else {
            EventAdapter.bt1.setBackground(Color.blue);
            EventAdapter.bt2.setBackground(Color.blue);
            EventAdapter.bt3.setBackground(Color.blue);
        }
    }
}
}

```

运行程序，单击标题为“绿色”的按钮，3 个按钮都变成绿色，运行结果如图 9.8 所示。



图 9.8 改变按钮的颜色

9.4 综合实例

在下面的综合实例中，将运用 AWT 的各种事件处理机制实现学生信息的注册和查看。在学生信息窗口中，单击“注册”按钮，弹出学生注册窗口。在学生注册窗口中填

入学生的学号和姓名，再单击“确定”按钮，完成该学生的信息注册；如果单击“取消”按钮，则取消该学生的注册信息。已经注册的学生，单击“查看”按钮时，其信息将出现在文本区域中。

这个程序灵活运用 AWT 的各种事件处理。这里，在“注册”按钮和“确定”按钮上注册类自身的监听器，在“查看”按钮上注册鼠标适配器，在“关闭”按钮和“取消”按钮上注册内部类监听器。

【例 9.7】 运用 AWT 的各种事件处理机制实现学生信息的注册和查看。

StudentInfo.java

```
package org.awt;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class StudentInfo extends Frame implements ActionListener{
    static HashMap map = new HashMap();
    Button bt1 = new Button();
    Button bt2 = new Button();
    TextArea textArea1 = new TextArea();
    Button bt3 = new Button();
    public StudentInfo() {
        try {
            initialize();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        StudentInfo student = new StudentInfo();
        student.setBackground(Color.cyan);
        student.setVisible(true);
        student.setSize(new Dimension(400, 250));           // 指定宽度和高度
        student.setTitle("学生信息");
    }
    private void initialize() throws Exception {
        this.setLayout(null);
        bt1.setLabel("注册");
        bt1.setBackground(Color.cyan);
        bt1.setBounds(new Rectangle(85, 200, 70, 25));
        bt1.addActionListener(this);                         // 本身实现监听接口
        bt2.setBounds(new Rectangle(170, 200, 70, 25));
        bt2.addMouseListener(new actionAdapter(this));      // 外部类实现监听接口
        bt2.setLabel("查看");
        bt2.setBackground(Color.cyan);
        textArea1.setEditable(false);
        textArea1.setText("");
        textArea1.setBackground(Color.white);
    }
}
```



```

        textArea1.setBounds(new Rectangle(80, 40, 250, 150));
        bt3.setLabel("关闭");
        bt3.setBackground(Color.cyan);
        /*******内部类实现监听接口*****
        bt3.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent evt){           // 关闭窗口
                System.exit(0);
            }
        });
        bt3.setBounds(new Rectangle(255, 200, 70, 25));
        this.setResizable(false);
        this.add(textArea1);
        this.add(bt2);
        this.add(bt1);
        this.add(bt3);
    }
    /*******实现监听器 ActionListener 的 actionPerformed 方法
    public void actionPerformed(ActionEvent e) {
        RegisterFrame regf = new RegisterFrame();
        regf.setTitle("信息注册");
        regf.setVisible(true);
        regf.setSize(new Dimension(350, 180));
        regf.setBackground(Color.cyan);
    }
}

class RegisterFrame extends Frame implements ActionListener {
    Label label1 = new Label();
    TextField textField1 = new TextField();
    Label label2 = new Label();
    TextField textField2 = new TextField();
    Button button1 = new Button();
    Button button2 = new Button();
    public RegisterFrame() {
        try {
            register();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void register() throws Exception {
        this.setLayout(null);
        label1.setText("学生学号: ");
        label1.setBackground(Color.cyan);
        label1.setBounds(new Rectangle(50, 50, 60, 20));
        textField1.setText("");
        textField1.setBounds(new Rectangle(120, 50, 140, 20));
        label2.setBounds(new Rectangle(50, 80, 60, 20));

```

```

        label2.setText("学生姓名: ");
        label2.setBackground(Color.cyan);
        textField2.setText("");
        textField2.setBounds(new Rectangle(120, 80, 140, 20));
        button1.setLabel("确定");
        button1.setBackground(Color.cyan);
        button1.setBounds(new Rectangle(85, 120, 70, 25));
        button1.addActionListener(this);
        button2.setBounds(new Rectangle(180, 120, 70, 25));
        button2.setBackground(Color.cyan);
        button2.addActionListener(new ActionListener() { // 内部类
            public void actionPerformed(ActionEvent evt){
                System.exit(0);
            }
        });
        button2.setLabel("取消");
        this.setResizable(false);
        this.add(button2,);
        this.add(label1);
        this.add(label2);
        this.add(textField2);
        this.add(textField1);
        this.add(button1);
    }

    /*******事件监听器*****
    public void actionPerformed(ActionEvent e) {
        String id = textField1.getText();
        String name = textField2.getText();
        StudentInfo.map.put(id, name);
        this.dispose();
    }
}

/*******鼠标事件适配器*****
class actionAdapter extends MouseAdapter{
    StudentInfo studentAdapter;
    public actionAdapter(StudentInfo studentInfo) {
        this.studentAdapter = studentInfo;
    }
    public void mouseClicked(MouseEvent evt) { // 鼠标单击事件
        Button button = (Button)evt.getSource();
        studentAdapter.textArea1.setText("");
        Iterator iter = studentAdapter.map.keySet().iterator(); // 遍历 map 容器
        while (iter.hasNext()) {
            String id = iter.next().toString();
            studentAdapter.textArea1.append("学号: " + id + "      姓名: "
                + studentAdapter.map.get(id).toString() + '\n');
        }
    }
}

```

```
}  
}
```

运行程序，单击“注册”按钮，出现学生信息注册窗口，如图 9.9 所示。在文本框中输入学生学号和学生姓名，单击“确定”按钮，完成学生信息的注册。



图 9.9 学生信息注册

多次单击“注册”按钮，完成多个学生的信息注册。单击“查看”按钮，已经注册的学生信息将出现在文本区域中，如图 9.10 所示。

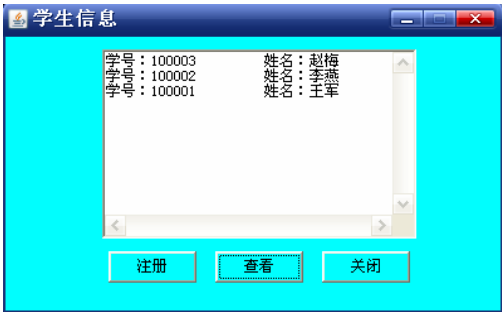


图 9.10 学生信息查看

第 10 章 Swing 组件及应用

Swing 组件库是 Java 推出的一套新的图形用户界面开发工具包，它简化了基于图形界面的窗口系统的开发。使用 Swing 可以方便地选择与设计自己需要的 GUI 风格，带来了更多的灵活性和更强大的功能。

Swing 是建立在 AWT 之上的，它利用了 AWT 的下层组件，包括图形、颜色、字体、布局管理器和工具包，也使用了 AWT 的事件处理机制，但基本上没有使用 AWT 的基本组件。Swing 提供了几十个组件，数倍于 AWT，增加了 Windows 环境下几乎所有的控件，极大地方便了图形用户界面的设计。Swing 组件几乎都是轻量组件，它采用了与 AWT 完全不同的工作方式，它将按钮、菜单这样的用户界面元素绘制在空白窗口上，而对等体只需要创建和绘制窗口。因此 Swing 的部件在程序运行的所有平台上的外观和动作都一样。

10.1 窗口 JFrame

JFrame 是带标题的顶层窗口，它继承 `java.awt.Frame`，但两者是有区别的。JFrame 不能直接通过 `add()` 方法加入组件，也不能直接通过 `setLayout()` 方法设置布局管理器。每个 JFrame 都有一个与之关联的内容面板（`contentPane`），只能针对这个 `contentPane` 设置布局管理器及加入组件。例如下面的代码片段：

```
JFrame jFrame = new JFrame();
Container contentPane = jFrame.getContentPane();
contentPane.setLayout(new GridLayout(2,1));
contentPane.add(jLabel);
contentPane.add(jButton);
jFrame.setContentPane(jPanel);
```

【例 10.1】 创建 JFrame 窗体，在窗体上添加两个 JLabel 和 JButton。

MyJFrame.java

```
package org.swing;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyJFrame extends JFrame{
    int i=0,j=0;
    JLabel jLabel1=new JLabel(" "+i,JLabel.CENTER);
    JLabel jLabel2=new JLabel(" "+j,JLabel.CENTER);
    public MyJFrame(){
        Container contentPane=this.getContentPane();
        contentPane.setLayout(new GridLayout(2,2)); // 使用网格布局管理器
        JButton jButton1=new JButton("ADD");
        JButton jButton2=new JButton("ADD");
```

```

jButton1.addActionListener(new ActionListener(){ // 注册监听器
    public void actionPerformed(ActionEvent e){
        i++;
        jLabel1.setText(" "+i);                //字符串值与整型值连接成为字符串值
    }
});
jButton2.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        j++;
        jLabel2.setfText(" "+j);
    }
});
contentPane.add(jLabel1);
contentPane.add(jLabel2);
contentPane.add(jButton1);
contentPane.add(jButton2);
this.setTitle("This is a JFrame");
this.setSize(300,300);
this.setVisible(true);
}
public static void main(String[] args){
    new MyJFrame().setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 关闭窗口体
}
}

```

程序运行结果如图 10.1 所示。

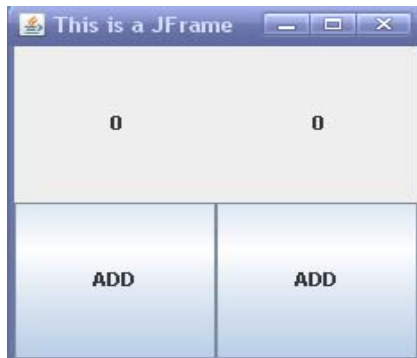


图 10.1 一个 JFrame

10.2 按钮

所有的按钮组件都继承自 `AbstractButton` 类，包括复选框（`JCheckBox`）、普通按钮（`Jbutton`）、单选按钮（`JradioButton`），甚至菜单项（`JmenuItem`）等。在按钮中可以显示图标，`ImageIcon` 类表示图标。`AbstractButton` 提供了以下和设置图标有关的方法。

- `setIcon(Icon icon)`: 设置按钮有效状态下的图标。

- `setRolloverIcon(Icon icon)`: 设置鼠标移动到按钮区域的图标。
- `setPressedIcon(Icon icon)`: 设置按下按钮时的图标。
- `setDisabledIcon(Icon icon)`: 设置按钮无效状态下的图标。

【例 10.2】 创建两个 `JButton`，让其中一个 `JButton` 在各个状态下使用不同的图标。

ButtonAndIcon.java

```
package org.swing;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonAndIcon extends JFrame {
    private static Icon[] icons;
    private JButton jbt1, jbt2 = new JButton("Disable");
    private boolean flag = false;
    public ButtonAndIcon(String title) {
        super(title);
        icons = new Icon[] {
            new ImageIcon(getClass().getResource("image0.jpg")),
            new ImageIcon(getClass().getResource("image1.jpg")),
            new ImageIcon(getClass().getResource("image2.jpg")),
            new ImageIcon(getClass().getResource("image3.jpg")),
            new ImageIcon(getClass().getResource("image4.jpg")), };
        jbt1 = new JButton("Pet", icons[0]);
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());           // 使用流式布局管理器
        jbt1.addActionListener(new ActionListener() {      // 注册监听器
            public void actionPerformed(ActionEvent e) {
                if (flag) {
                    jbt1.setIcon(icons[0]);
                    flag = false;
                } else {
                    jbt1.setIcon(icons[1]);
                    flag = true;
                }
                jbt1.setVerticalAlignment(JButton.TOP);
                jbt1.setHorizontalAlignment(JButton.LEFT);
            }
        });
        jbt1.setRolloverEnabled(true);
        jbt1.setRolloverIcon(icons[2]);
        jbt1.setPressedIcon(icons[3]);
        jbt1.setDisabledIcon(icons[4]);
        jbt1.setToolTipText("Click Me!");
        contentPane.add(jbt1);
        jbt2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (jbt1.isEnabled()) {
```

```

        jbt1.setEnabled(false);                // 使按钮失效
        jbt2.setText("Enable");
    } else {
        jbt1.setEnabled(true);                // 使按钮有效
        jbt2.setText("Disable");
    }
    }
});
contentPane.add(jbt2);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 关闭窗口体
pack();
setVisible(true);
}
public static void main(String[] args) {
    new ButtonAndIcon("use Buttons");
}
}

```

说明：程序的 `getClass().getResource("image0.jpg")` 方法从当前路径下加载图片文件，程序运行结果如图 10.2 所示。图中共有两个按钮，Pet 按钮(jbt1)显示图标，Disable 按钮(jbt2)能够控制 jbt1 按钮是否有效。



图 10.2 Button 的使用

10.3 JTextField与JTextArea

文本框是具有输入单行文本和编辑功能的组件。把文本框添加到窗口中的常用方法是把它添加到面板或其他容器中，例如下面的代码片段：

```

JPanel jPanel = new JPanel();
JTextField text = new JTextField("default input",20);
jPanel.add(text);

```

这段代码将添加一个文本框，同时通过放入一个字符串“default input”来对它进行初始化，构造器的第二个参数设置文本框的宽度为 20。

相对 `JTextField`，`JTextArea` 可以输入多行文本。当在 `JTextField` 内输入回车时，将触发 `ActionEvent` 事件；但当在 `JTextArea` 中输入回车时，仅仅意味着换行输入文本，并不会触发

ActionEvent 事件，应该使用另外的按钮或菜单来触发 ActionEvent 事件。JScrollPane 表示带滚动条的面板，在默认的情况下，只有当面板的内容超过了面板的面积时，才会显示滚动条。

【例 10.3】 求 a 到 b 之间的所有质数，每行显示 c 个。 a 、 b 、 c 的值由单行编辑框输入，结果在 多行编辑框上显示，并显示质数个数。

TextAreaDemo.java

```
package org.swing;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TextAreaDemo extends JFrame {
    static JTextField tf1 = new JTextField();
    static JTextField tf2 = new JTextField();
    static JTextField tf3 = new JTextField();
    static TextArea ta = new TextArea();
    static JTextField tf4 = new JTextField();
    static int num = 0;
    //*****显示滚动条*****
    JScrollPane jp = new JScrollPane(ta,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

    public TextAreaDemo() {
        setBackground(Color.cyan);
        Container contentPane = getContentPane();
        contentPane.setLayout(null); // 取消布局管理器
        contentPane.setBackground(Color.cyan); // 设置窗口的颜色
        contentPane.add(jp);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 关闭窗口
        setLocation(300, 300);
        setSize(new Dimension(500, 400));
        Button bt1 = new Button("求 a 到 b 之间的质数");
        bt1.setBackground(Color.cyan);
        bt1.addActionListener(new GetAction()); // 注册事件监听器
        Button bt2 = new Button("质数个数");
        bt2.setBackground(Color.cyan);
        JLabel l1 = new JLabel("输入 a 的值"); // 创建一个标签
        JLabel l2 = new JLabel("输入 b 的值");
        JLabel l3 = new JLabel("每行显示个数");
        tf1.setBounds(new Rectangle(40, 50, 70, 25));
        tf2.setBounds(new Rectangle(130, 50, 70, 25));
        tf3.setBounds(new Rectangle(220, 50, 70, 25));
        ta.setEditable(true);
        ta.setText("");
        ta.setBackground(Color.white); // 设置文本区的颜色
        ta.setBounds(new Rectangle(40, 100, 400, 200));
        l1.setBounds(new Rectangle(40, 20, 60, 25));
        l2.setBounds(new Rectangle(130, 20, 60, 25));
        l3.setBounds(new Rectangle(220, 20, 60, 25));
```



```

l3.setBounds(new Rectangle(220, 20, 120, 25));
bt1.setBounds(new Rectangle(340, 20, 120, 25));
bt2.setBounds(new Rectangle(40, 330, 50, 25));
tf4.setBounds(new Rectangle(130, 330, 70, 25));
contentPane.add(l1); // 加入组件
contentPane.add(l2);
contentPane.add(l3);
contentPane.add(bt1);
contentPane.add(bt2);
contentPane.add(ta);
contentPane.add(tf1);
contentPane.add(tf2);
contentPane.add(tf3);
contentPane.add(tf4);
setVisible(true); // 使窗口可见
}

public static void main(String[] args) {
    TextAreaDemo test = new TextAreaDemo();
}

}

class GetAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String text1 = TextAreaDemo.tf1.getText(); // 获取文本框中的内容
        String text2 = TextAreaDemo.tf2.getText();
        String text3 = TextAreaDemo.tf3.getText();
        int a, b, c;
        a = Integer.parseInt(text1); // 将字符串类型转换为整型
        b = Integer.parseInt(text2);
        c = Integer.parseInt(text3);
        boolean flag;
        int m, p, count = 0;
        for (m = a; m <= b; m++) {
            flag = true;
            for (p = 2; p <= m / 2; p++)
                if (m % p == 0) { // 判断是否是质数
                    flag = false;
                    break;
                }
            if (flag) {
                String str = String.valueOf(m); // 将整型转换为字符串类型
                TextAreaDemo.ta.append(str + " "); // 将质数写入到文本区中
                count++;
                TextAreaDemo.num++;
                if (count % c == 0) { // 每行中只输出 c 个质数
                    TextAreaDemo.ta.append("\n");
                }
            }
        }
    }
}

```

```

    }
    String str = String.valueOf(TextAreaDemo.num);
    TextAreaDemo.tf4.setText(str);
}
}

```

运行程序，在文本框中依次输入“5、500、7”，单击“求 a 到 b 之间的质数”按钮，将在文本区中显示 5~500 之间的所有质数，并计算出质数个数，显示在下面的文本框中，程序运行结果如图 10.3 所示。



图 10.3 计算 5~500 之间的质数

10.4 JCheckBox和JRadioButton

JCheckBox 表示复选框，用户可以同时选择多个选项。JRadioButton 表示单选按钮，可以把多个单选按钮加入到一个按钮组（ButtonGroup），在任何时候，用户只能选择按钮组中的一个按钮。当用户选择了一个单选按钮时，将触发 ActionEvent 事件，由 ActionListener 来处理。

【例10.4】 创建一个复选按钮和一个按钮组，按钮组中有多个单选按钮。选择的内容可在列表框中显示。

MyCheckBox.java

```

package org.swing;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
class Favorite extends JPanel {
    JCheckBox jCheck1, jCheck2, jCheck3, jCheck4;
    Favorite() {
        jCheck1 = new JCheckBox("运动");
        jCheck2 = new JCheckBox("电脑");
        jCheck3 = new JCheckBox("音乐");

```

```

        jCheck4 = new JCheckBox("读书");
        add(new JLabel("爱好"));           // 把标签加载到 JPanel 上
        add(jCheck1);                     // 把 JCheckBox 加载到 JPanel 上
        add(jCheck2);
        add(jCheck3);
        add(jCheck4);
    }
}

class SexBox extends JPanel {
    JRadioButton jRadio1, jRadio2;
    SexBox() {
        jRadio1 = new JRadioButton("男");
        jRadio2 = new JRadioButton("女");
        add(new JLabel("性别"));           // 把标签加载到 JPanel 上
        ButtonGroup bg = new ButtonGroup();
        bg.add(jRadio1);
        bg.add(jRadio2);
        add(jRadio1);
        add(jRadio2);
    }
}

class NameBox extends JPanel {
    JTextField jText;
    NameBox() {
        jText = new JTextField(10);
        add(new JLabel("姓名"));
        add(jText);
    }
}

class ThreeButton extends JPanel {
    JButton jButton1, jButton2, jButton3;
    ThreeButton() {
        jButton1 = new JButton("List");
        jButton2 = new JButton("Save");
        jButton3 = new JButton("Exit");
        add(jButton1);
        add(jButton2);
        add(jButton3);
    }
}

class MyCheckBox extends JFrame implements ActionListener {
    Favorite favorite;
    SexBox sex;
    NameBox name;
    JTextArea JText;
    ThreeButton tb;
    MyCheckBox() {

```

```

super("调查表");
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container container = this.getContentPane();
container.setLayout(new FlowLayout());
testInit();
container.add(name);
container.add(sex);
container.add(favorite);
container.add(new JScrollPane(JText));
container.add(tb);
this.setBounds(300, 200, 280, 300);
this.setVisible(true);
}
void testInit() {
    favorite = new Favorite ();
    sex = new SexBox();
    name = new NameBox();
    JText = new JTextArea(5, 22);
    tb = new ThreeButton();
    tb.jButton1.addActionListener(this);           // 注册自身的监听器
    tb.jButton2.addActionListener(this);
    tb.jButton3.addActionListener(this);
}
public void actionPerformed(ActionEvent e) {
    Object o = e.getSource();
    if (o == tb.jButton1) {
        StringBuffer ss = new StringBuffer("\n 姓名: " + name.jText.getText()
            + "\n 性别: ");
        if (sex.jRadio1.isSelected() == true)
            ss.append("男");
        else if (sex.jRadio2.isSelected() == true)
            ss.append("女");
        ss.append("\n 爱好: ");
        if (favorite.jCheck1.isSelected() == true)
            ss.append("运动");
        if (favorite.jCheck2.isSelected() == true)
            ss.append("电脑");
        if (favorite.jCheck3.isSelected() == true)
            ss.append("音乐");
        if (favorite.jCheck4.isSelected() == true)
            ss.append("读书");
        JText.setText(ss.toString());
    }
    else if (o == tb.jButton2) {
        try {
            FileWriter out = new FileWriter("D:\\temp.txt", true);
            out.write("\r\n" + JText.getText());

```

```

        JOptionPane.showMessageDialog(this, "文件已保存!");
        out.close();
    } catch (Exception ex) {
    }
}
else
    System.exit(0);
}
public static void main(String[] args) {
    new MyCheckBox();
}
}

```

运行该程序，在文本框中输入姓名，选择性别和爱好，单击“List”按钮，则刚才操作的信息将显示在列表框中，单击“保存”按钮信息将保存到文件中，如图 10.4 所示。



图 10.4 组合界面

10.5 菜单条 (JMenuBar)

菜单的组织方式为：一个菜单条 JMenuBar 中可以包含多个菜单 JMenu，一个菜单 JMenu 中可以包含多个菜单项 JMenuItem。有一些支持菜单的组件如 JFrame、JDialog 都有一个 setJMenuBar (JMenuBar bar) 方法，可以用这个方法设置菜单条。

【例 10.5】 创建两个菜单、多个菜单项。当选择某一菜单项时，在窗口中显示不同的卡片，同时在窗口底部显示所选的菜单项。当选择“状态栏”菜单时，窗口底部的显示标签可见，否则不可见。有的菜单项加了快捷键，如“Shift+O”、“Ctrl+Shift+S”、“Ctrl+X”等。

JMenTest.java

```

package org.swing;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JMenuTest {
    JFrame f = new JFrame("Swing 菜单的用法");
    JLabel stat = new JLabel("这里是状态栏");
}

```

```

Font ft = new Font("Serif", Font.BOLD, 18);
JLabel l1 = new JLabel("这里是西方", JLabel.CENTER);
JLabel l2 = new JLabel("这里是中央", JLabel.CENTER);
JLabel l3 = new JLabel("这里是东方", JLabel.CENTER);
JPanel pc = new JPanel();
CardLayout c = new CardLayout();           // 创建一个布局管理器 CardLayout 的对象 c
JMenuBar menubar1 = new JMenuBar();        // 创建一个菜单条
JMenu menu1 = new JMenu("视图");           // 定义一个菜单对象 menu1，其标题为“视图”
JMenu menu2 = new JMenu("编辑");
// 定义一个菜单项 JMenuItem 的对象 mitm1，其标题为“西方”
JMenuItem mitm1 = new JMenuItem("西方");
JMenuItem mitm2 = new JMenuItem("中央");
JMenuItem mitm3 = new JMenuItem("东方");
JMenuItem mitm4 = new JMenuItem("剪下");
JMenuItem mitm5 = new JMenuItem("粘贴");
// 定义一个菜单项 JCheckBoxMenuItem 的对象 mitm6，其标题为“状态栏”，选中
JCheckBoxMenuItem mitm6 = new JCheckBoxMenuItem("状态栏", true);
JMenuItem mitm7 = new JMenuItem("退出");
public static void main(String args[]) {
    JMenuTest that = new JMenuTest();
    that.go();
}
public void go() {
    f.setSize(350, 300);
    menubar1.add(menu1);                    // 添加 menu1 到 MenuBar 中
    menubar1.add(menu2);
    menu1.add(mitm1);
    menu1.add(mitm2);
    // 为菜单项 mitm1 添加快捷键 Shift+O
    mitm1.setAccelerator(KeyStroke.getKeyStroke('O', KeyEvent.SHIFT_MASK, false));
    mitm2.setAccelerator(KeyStroke.getKeyStroke('S', KeyEvent.CTRL_MASK
        + KeyEvent.SHIFT_MASK, false));    // 为菜单项 mitm2 添加快捷键 Ctrl+Shift+S
    menu1.add(mitm3);
    menu1.addSeparator();                  // 添加一条分隔线
    menu1.add(mitm6);
    menu1.addSeparator();
    menu1.add(mitm7);
    mitm7.setAccelerator(KeyStroke.getKeyStroke('X', KeyEvent.CTRL_MASK,
        false));                          // 为菜单项 mitm7 添加快捷键 Ctrl+X
    menu2.add(mitm4);
    menu2.add(mitm5);
    f.setJMenuBar(menubar1);              // 设定窗口 f 的菜单条为 menubar1
    f.getContentPane().add("Center", pc);  // 将容器 pc 加到窗口 f 的中央
    f.getContentPane().add("South", stat); // 将标签 stat 加到窗口 f 的底部
    pc.setLayout(c);
    pc.add(l1, "west");
    pc.add(l2, "center");
}

```

```

pc.add(l3, "east");
// 将菜单项 mitm1 注册到监听器 JMenuHandler 上, 参数 1 代表 mitm1
mitm1.addActionListener(new JMenuHandler(1));
mitm2.addActionListener(new JMenuHandler(2));
mitm3.addActionListener(new JMenuHandler(3));
mitm4.addActionListener(new JMenuHandler(4));
mitm5.addActionListener(new JMenuHandler(5));
mitm7.addActionListener(new JMenuHandler(7));
// JCheckBoxMenuItem 不响应(ActionEvent) 事件, 这里用 ItemEvent 事件
mitm6.addItemListener(new JMenuDisp());
f.addWindowListener(new WinHandler());
l1.setFont(ft); // 设置菜单字体
l2.setFont(ft);
l3.setFont(ft);
stat.setFont(ft);
menu1.setFont(ft);
menu2.setFont(ft);
mitm1.setFont(ft);
mitm2.setFont(ft);
mitm3.setFont(ft);
mitm4.setFont(ft);
mitm5.setFont(ft);
mitm6.setFont(ft);
mitm7.setFont(ft);
f.setVisible(true);
}

class JMenuDisp implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        //若菜单项被选择, 即前面有一个标记, 则将标签 stat 置为可见, 否则置为不可见
        if (mitm6.getState())
            stat.setVisible(true);
        else
            stat.setVisible(false);
    }
}

class JMenuHandler implements ActionListener {
    private int ch;

    JMenuHandler(int select) {
        ch = select;
    }

    public void actionPerformed(ActionEvent e) {
        switch (ch) {
            case 1:
                c.show(pc, "west"); // 若选择了 mitm1, 则显示名为 west 的卡片
                break;
            case 2:
                c.show(pc, "center");

```

```

        break;
    case 3:
        c.show(pc, "east");
        break;
    case 4:
    case 5:
        break;
    case 7:
        System.exit(-1);
    }
    stat.setText("你选择的菜单项是: " + e.getActionCommand());
}
}

class WinHandler extends WindowAdapter {
    public void windowClosing(WindowEvent e) { // 关闭窗口
        System.exit(-1);
    }
}
}

```

程序的运行结果如图 10.5 所示。



图 10.5 一个 JMenuBar

10.6 弹出式菜单 (JPopupMenu)

从名称中就可看出, JPopupMenu 并不固定在菜单栏中, 而是能够自由浮动。JPopupMenu 具有很好的环境相关特性, 每一个 JPopupMenu 都与相应的控件项关联, 该控件被称做调用者。下面的代码创建一个带有标题的 JPopupMenu。

```
JPopupMenu myJPopupMenu = new JPopupMenu("菜单");
```

可以使用 add() 方法或 insert() 方法向 JPopupMenu 中添加或者插入 JMenuItem 与 JComponent。JPopupMenu 对添加到其中的每一个菜单项都赋予一个整数索引, 并根据弹出式菜单的布局管理器调整菜单项显示的顺序。此外, 还可以使用 addSeparator() 方法添加分割线, 并且 JPopupMenu 也会为该分割线指定一个整数索引。若鼠标事件是平台的弹出式菜单触发事件, 则调用弹出式菜单对象的 show 方法来显示弹出式菜单。下面的 showJPopupMenu 方法在收到触发器事件时就会显示弹出式菜单, 代码如下。


```

public void showJPopupMenu(MouseEvent e) {
    if (e.isPopupTrigger()) {    // 若鼠标事件是平台的弹出式菜单，则触发事件
        myJPopupMenu.show(invoker,e.getX(),e.getY());
    }
}

```

弹出式菜单事件的判断，建议放在鼠标按下（mousePressed）及释放（mouseReleased）中进行。

【例 10.6】 创建一个弹出式菜单。

MyJPopupMenu.java

```

package org.swing;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyJPopupMenu extends JFrame {
    JMenu fileMenu;
    JPopupMenu jPopupMenuOne;
    JMenuItem openFile, closeFile, exit;
    JRadioButtonMenuItem copyFile, pasteFile;
    ButtonGroup buttonGroupOne;

    public MyJPopupMenu() {
        jPopupMenuOne = new JPopupMenu();    // 创建 jPopupMenuOne 对象
        buttonGroupOne = new ButtonGroup();
        // 创建文件菜单及子菜单，并将子菜单添加到文件菜单中
        fileMenu = new JMenu("文件");
        openFile = new JMenuItem("打开");
        closeFile = new JMenuItem("关闭");
        fileMenu.add(openFile);
        fileMenu.add(closeFile);
        jPopupMenuOne.add(fileMenu);    // 将 fileMenu 菜单添加到弹出式菜单中
        jPopupMenuOne.addSeparator();    // 添加分隔符
        copyFile = new JRadioButtonMenuItem("复制");
        pasteFile = new JRadioButtonMenuItem("粘贴");
        buttonGroupOne.add(copyFile);
        buttonGroupOne.add(pasteFile);
        jPopupMenuOne.add(copyFile);    // 将 copyFile 添加到 jPopupMenuOne 中
        jPopupMenuOne.add(pasteFile);    // 将 pasteFile 添加到 jPopupMenuOne 中
        jPopupMenuOne.addSeparator();
        exit = new JMenuItem("退出");
        jPopupMenuOne.add(exit);    // 将 exit 添加到 jPopupMenuOne 中
        MouseListener popupListener = new PopupListener(jPopupMenuOne);
        this.addMouseListener(popupListener);    // 向主窗口注册监听器
        this.setTitle("This is a JPopupMenu");
        this.setBounds(100, 100, 250, 150);
        this.setVisible(true);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

```

```

public static void main(String args[]) {
    new MyJPopupMenu();
}
//*****添加内部类，它扩展了 MouseAdapter 类，用来处理鼠标事件*****
class PopupListener extends MouseAdapter {
    JPopupMenu popupMenu;
    PopupListener(JPopupMenu popupMenu) {
        this.popupMenu = popupMenu;
    }
    public void mousePressed(MouseEvent e) {
        showPopupMenu(e);
    }
    public void mouseReleased(MouseEvent e) {
        showPopupMenu(e);
    }
    private void showPopupMenu(MouseEvent e) {
        if (e.isPopupTrigger()) {
            popupMenu.show(e.getComponent(), e.getX(), e.getY());
        }
    }
}
}

```

运行程序，右击鼠标显示弹出式菜单，如图 10.6 所示。

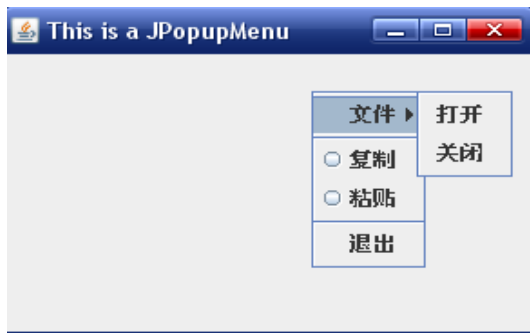


图 10.6 弹出式菜单界面

10.7 综合实例

在这个综合实例中，设计一个具备基本功能的计算器。

思路：程序创建一个窗体和 Panel。在 Panel 中的上方添加一个 JButton 按钮，用于显示要计算的值和计算后的结果，在 Panel 中的下方添加 16 个 JButton 按钮。10 个按钮是数字 0～9，其余则是运算符和小数点。通过单击 Panel 中下方的数字按钮和命令按钮进行数字运算，这就要求在这些计算器按钮上注册相应的事件监听器。在数字按钮上注册事件监听器用于获取相应的数值，在命令按钮上注册事件监听器用于获取运算符，从而计算结果。

MyFrame.java

```
package org.swing;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MyFrame extends JFrame {
    public MyFrame() {
        setTitle("Calculator");
        MyPanel panel = new MyPanel();
        add(panel);
        pack(); // 调整此计算器窗口的大小
    }
}

class MyPanel extends JPanel {
    private JButton display; // 显示单击计算器按钮的值和计算后的值
    private JPanel panel; // 计算器面板
    private double result;
    private String lastCommand; // 计算器的命令按钮
    private boolean start; // 是否开始计算
    public MyPanel() {
        setLayout(new BorderLayout());
        result = 0;
        lastCommand = "=";
        start = true;
        display = new JButton("0");
        display.setEnabled(false);
        add(display, BorderLayout.NORTH);
        ActionListener insert = new InsertAction();
        ActionListener command = new CommandAction();
        // 以网格布局管理器管理 16 个计算器按钮
        panel = new JPanel();
        panel.setLayout(new GridLayout(4, 4)); // 在 JPanel 布局 16 个计算器按钮
        addButton("7", insert); // 把计算器按钮加到计算器面板上
        addButton("8", insert); addButton("9", insert);
        addButton("/", command); addButton("4", insert);
        addButton("5", insert); addButton("6", insert);
        addButton("*", command); addButton("1", insert);
        addButton("2", insert); addButton("3", insert);
        addButton("-", command); addButton("0", insert);
        addButton(".", insert); addButton("=", command);
        addButton("+", command);
        add(panel, BorderLayout.CENTER);
    }
    //*****添加计算器按钮到计算器面板上*****
    private void addButton(String label, ActionListener listener) {
```

```

        JButton button = new JButton(label);
        button.addActionListener(listener);           // 注册事件监听器
        panel.add(button);
    }
    // 设置 display 的值为所单击的计算器按钮的值
    private class InsertAction implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            // 返回与此动作相关的命令字符串
            String input = event.getActionCommand();
            if (start) {
                display.setText("");
                start = false;
            }
            display.setText(display.getText() + input); // 显示单击的计算器按钮的值
        }
    }
    //*****依单击计算器上的内容执行命令*****
    private class CommandAction implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            String command = event.getActionCommand();
            if (start) {
                if (command.equals("-")) {
                    display.setText(command); // 如果单击的是“-”按钮，说明是负数
                    start = false;
                } else
                    lastCommand = command;
            } else {
                // 把字符串转换为 Double 类型并计算结果
                calculate(Double.parseDouble(display.getText()));
                lastCommand = command;
                start = true;
            }
        }
    }
    // *****计算结果*****
    public void calculate(double x) {
        if (lastCommand.equals("+"))    result += x;
        else if (lastCommand.equals("-")) result - = x;
        else if (lastCommand.equals("*")) result *= x;
        else if (lastCommand.equals("/")) result /= x;
        else if (lastCommand.equals("=")) result = x;
        display.setText("" + result);
    }
}

public class Calculator {
    public static void main(String[] args) {
        MyFrame frame = new MyFrame();
    }
}

```

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);    // 关闭计算器窗口
frame.setVisible(true);

    }
}
```

运行程序，单击任意数值和运算符号，计算结果，如图 10.7 所示。



图 10.7 一个计算器

第 11 章 并 发

进程是指运行中的应用程序，每一个进程都有自己独立的内存空间。一个应用程序可以同时启动多个进程。例如，每次运行 JDK 的 `java.exe` 程序，就启动了一个独立的 Java 虚拟机进程，该进程的任务是解析并运行 Java 程序代码。

线程是指进程中能够独立执行的控制流。一个进程可以由多个线程组成，即在一个进程中可以同时运行多个不同的线程，它们分别执行不同的任务。当进程的多个线程同时运行时，这种运行方式称为并发运行。许多服务器程序，如数据库服务器和 Web 服务器，都支持并发运行，这些服务器能同时响应来自不同客户的请求。Java 虚拟机中的线程并发运行，可以由拥有多处理器的硬件支持，也可以由拥有单处理器的硬件轮流支持。

线程和进程的主要区别在于：每个进程都需要操作系统为其分配独立的内存地址空间，而同一进程中的所有线程在同一块地址空间中工作。这些线程可以共享进程的状态和资源，比如共享一个对象或者共享已经打开的一个文件。

11.1 线程的创建与启动

每当用 `java` 命令启动一个 Java 虚拟机进程时，Java 虚拟机都会创建一个主线程，该线程从程序入口 `main()` 方法开始运行。用户也可以创建 `Thread` 的实例来创建新的线程，它将与主线程并发运行。创建线程有两种方式：继承 `java.lang.Thread` 类和实现 `Runnable` 接口。

11.1.1 继承 `java.lang.Thread` 类

`Thread` 类代表线程类，它的常用方法如下。

- `static Thread currentThread()`: 返回当前正在运行的线程对象的引用。
- `static void yield()`: 暂停当前正在运行的线程对象，并运行其他线程。
- `static void sleep(long millis) throws InterruptedException`: 在指定的毫秒数内让当前正在运行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。该线程不执行释放对象锁的操作。`millis` 表示以毫秒为单位的休眠时间。
- `void start()`: 使该线程开始运行，Java 虚拟机调用该线程的 `run()` 方法，该线程与其他线程并发执行。
- `void run()`: 如果该线程是使用实现 `Runnable` 接口的对象构造的，则调用该 `Runnable` 对象的 `run()` 方法；否则，该方法不执行任何操作并返回。`Thread` 的子类应该重写该方法。
- `void interrupt()`: 设置线程的中断标记位，请求线程停止运行。
- `final void setPriority(int newPriority)`: 更改线程的优先级。
- `final int getPriority()`: 返回线程的优先级。
- `final void setName(String name)`: 改变线程名称，使之与参数 `name` 相同。
- `final String getName()`: 返回该线程的名称。

- `final void join()throws InterruptedException`: 等待该线程终止。
- `final void setDaemon(boolean on)`: 将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时, Java 虚拟机退出。该方法必须在启动线程前调用。
- `long getId()`: 返回该线程的标识符。线程 ID 是一个正的长整型数, 在创建该线程时生成。线程 ID 是唯一的, 在线程终止之前一直保持不变。线程终止后, 该线程 ID 可以被重新使用。

用户的线程类只需要继承 `Thread` 类, 并重写 `Thread` 类的 `run()`方法即可。通过调用用户线程类的 `start()`方法即可启动该线程。

【例 11.1】 继承 `Thread` 类实现多线程。

TestThread.java

```
package org.concurrency.expansion;
class MyThread extends Thread {
    private int a = 0;
    public void run() {
        for (int a = 0; a < 10; a++) {
            System.out.println(currentThread().getName() + ":" + a);
            try {
                sleep(100);           // 给其他线程运行的机会
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

public class TestThread {
    public static void main(String[] args) {
        MyThread thread = new MyThread();    // 创建用户线程对象
        thread.start();                      // 启动用户线程
        thread.run();                        // 主线程调用用户线程对象的 run()方法
    }
}
```

程序的一种可能运行结果:

```
main:0
Thread-0:0
main:1
Thread-0:1
main:2
Thread-0:2
Thread-0:3
main:3|
main:4
Thread-0:4
Thread-0:5
main:5
main:6
Thread-0:6
Thread-0:7
main:7
main:8
Thread-0:8
Thread-0:9
main:9
```

说明 :当运行 `java TestThread` 命令时, Java 虚拟机首先创建并启动主线程。主线程从 `main()`

方法开始运行。主线程创建一个用户线程并启动用户线程。

`Thread` 类的 `currentThread()` 静态方法返回当前线程的引用, `Thread` 类的 `getName()` 方法则返回线程的名字。每个线程都有默认的名字, 主线程默认的名字为 “main”。用户创建的第一个线程的默认的名字为 “Thread-0”, 第二个线程的默认名字为 “Thread-1”, 依次类推。为了让每个线程能轮流获得 CPU, 在 `run()` 方法中还调用了 `Thread` 类的 `sleep()` 静态方法, 该方法让当前线程放弃 CPU 并且睡眠若干时间。

11.1.2 实现 `Runnable` 接口

另一种创建线程的方式是实现 `java.lang.Runnable` 接口, 它只有一个 `run()` 方法。当使用 `Thread(Runnable thread)` 方式创建线程对象时, 需要为该方法传递一个实现了 `Runnable` 接口的对象, 这样创建的线程将调用实现 `Runnable` 接口的对象的 `run()` 方法。

【例 11.2】 实现 `Runnable` 接口实现多线程。

TestThread1.java

```
package org.concurrency.expansion;
public class TestThread1 {
    public static void main(String args[]) {
        MyThread1 mt = new MyThread1();
        Thread t = new Thread(mt);           // 创建用户线程
        t.start();                           // 启动用户线程
        for (int a = 0; a < 10; a++) {
            System.out.println(Thread.currentThread().getName() + ":" + a);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

class MyThread1 implements Runnable {      // 通过实现 Runnable 接口来创建线程
    public void run() {
        for (int a = 0; a < 10; a++) {
            System.out.println(Thread.currentThread().getName() + ":" + a);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```


程序的一种可能运行结果:

```
main:0
Thread-0:0
main:1
Thread-0:1
Thread-0:2
main:2
main:3
Thread-0:3
Thread-0:4
main:4
Thread-0:5
main:5
main:6
Thread-0:6
Thread-0:7
main:7
main:8
Thread-0:8
Thread-0:9
main:9
```

直接继承 `Thread` 类和实现 `Runnable` 接口都能实现多线程，在实际使用时究竟使用哪一个可参考如下内容。

如果只是为了实现 `Thread` 的执行过程，那么没有必要从 `Thread` 中派生。这是因为实现 `Runnable` 接口的对象代表的是一个计算任务，这个计算任务通常交由其他线程（如线程池中的线程）去执行。因此，`Runnable` 对应要完成的任务，`Thread` 对应任务的执行者。另外，若一个类已经有了父类，那么只能实现 `Runnable` 接口来参与多线程的运行；如果要使用并扩展 `Thread` 类的功能，那么可选择从 `Thread` 类继承。

11.2 线程的状态转换

线程在它的生命周期中会处于各种状态：新建、就绪、阻塞和死亡状态。

1. 新建状态 (New)

用 `new` 操作符创建一个线程对象时，例如用 `new Thread (t)`，线程还没有开始运行，此时线程处在新建（new）状态。当一个线程处在新建状态时，程序还没有开始运行线程中的代码。在线程可以运行之前，还有一些工作要做。

2. 就绪状态 (Runnable)

当一个线程对象创建后，其他线程调用它的 `start()` 方法，该线程就进入就绪状态。在这种状态下，只要调度器把时间片分配给线程，线程就可以运行。也就是说，在任意时刻，线程可以运行也可以不运行，这不同于死亡状态和阻塞状态。`Java` 虚拟机为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得 `CPU` 的使用权。`Java` 不区分就绪状态和运行状态。

3. 阻塞状态 (Blocked)

阻塞状态是指线程因为某些原因放弃 `CPU`，暂时停止运行。当线程处于阻塞状态时，`Java` 虚拟机不会给线程分配 `CPU`，直到线程重新进入就绪状态，它才有机会得到运行。例如，一个线程使用同步 `socket` 操作读取网络数据，而恰恰此时网络繁忙，或者远方主机无法响应，那么该线程必定会由于等待远端主机发送数据包而进入阻塞状态；或者，在线程体内调用了

同步对象的 wait()方法，线程也会进入阻塞状态。

一个线程进入阻塞状态可能有如下原因。

- (1) 通过调用 sleep(milliseconnds)使线程进入休眠状态，线程在指定的时间内不会运行。
- (2) 通过调用 wait()方法使线程挂起，直到线程得到了 notify()或 notifyAll()消息，线程才会进入就绪状态。
- (3) 线程在等待某个输入、输出完成。
- (4) 线程试图在某个对象上调用同步控制方法，但是对象锁不可用，因为另一个对象已经获取了这个锁。

【例 11.3】 调用 sleep()方法使主线程睡眠，进入阻塞状态，让客户线程得到 CPU。

TestBlocked.java

```
import java.util.*;
public class TestBlocked {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
        try {
            Thread.sleep(10000);           // 主线程睡眠 10s
        } catch (InterruptedException e) {
        }
        thread.interrupt();                // 中断客户线程
    }
}
class MyThread extends Thread {
    boolean flag = true;
    public void run() {
        while (flag) {
            System.out.println("...." + new Date() + "...");
            try {
                sleep(1000);
            } catch (InterruptedException e) {    // 接收到中断请求，立即结束 run()方法
                return;
            }
        }
    }
}
```

程序运行结果：

```
....Sat Mar 28 09:41:19 CST 2009...
....Sat Mar 28 09:41:21 CST 2009...
....Sat Mar 28 09:41:22 CST 2009...
....Sat Mar 28 09:41:23 CST 2009...
....Sat Mar 28 09:41:24 CST 2009...
....Sat Mar 28 09:41:25 CST 2009...
....Sat Mar 28 09:41:26 CST 2009...
....Sat Mar 28 09:41:27 CST 2009...
....Sat Mar 28 09:41:28 CST 2009...
....Sat Mar 28 09:41:29 CST 2009...
```

4. 死亡状态 (Dead)

当线程退出 `run()` 方法时，就进入死亡状态，该线程结束生命周期。线程有可能是正常执行完 `run()` 方法而退出，也有可能是遇到异常而退出 `run()` 方法。

11.3 线程调度

多线程的并发运行是指线程轮流获得 CPU 的使用权，分别执行各自的任务或在多处理器硬件上并行执行。在可运行池中，会有多个处于就绪状态的线程在等待 CPU，Java 虚拟机的一项任务就是负责线程的调度。线程的调度是指按照特定机制为多个线程分配 CPU 的使用权。有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有线程轮流获得 CPU 的使用权，并且平均分配每个线程占用 CPU 的时间片。Java 虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用 CPU，如果可运行池中线程的优先级相同，那么就随机选择一个线程，使其占用 CPU。处于运行状态的线程一直运行，直至它放弃 CPU。

如果希望明确地让一个线程给另一个线程运行的机会，可以采取以下办法之一。

- 调整各个线程的优先级。
- 让处于运行状态的线程调用 `Thread.sleep()` 方法。
- 让处于运行状态的线程调用 `Thread.yield()` 方法。
- 让处于运行状态的线程调用另一个线程的 `join()` 方法。

11.3.1 调整各个线程的优先级

所有处于就绪状态的线程根据优先级存放在可运行池中，优先级低的线程获得较少的运行机会，优先级高的线程获得较多的运行机会。`Thread` 类的 `setPriority(int)` 和 `getPriority()` 方法分别用来设置优先级和读取优先级。优先级用整数表示，取值范围是 1~10。`Thread` 类有以下 3 个静态常量。

- `MAX_PRIORITY`：取值为 10，表示最高优先级。
- `MIN_PRIORITY`：取值为 1，表示最低优先级。
- `NORM_PRIORITY`：取值为 5，表示默认的优先级。

【例 11.4】 调整线程的优先级，调整线程对象 `t2` 的优先级为最高级。

TestPriority.java

```
import java.util.Date;
public class TestPriority {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread1());
        Thread t2 = new Thread(new MyThread2());
        t2.setPriority(Thread.MAX_PRIORITY);           // 设置线程对象 t2 为最高级
        t1.start();                                   // 启动线程 t1
        t2.start();
    }
}
class MyThread1 extends Thread {                    // 通过继承 Thread 类来创建线程类
```

```

public void run() {
    //*****获取线程的优先级*****
    for (int i = 0; i < 10; i++) {
        System.out.println(currentThread().getName() + ":" + currentThread().getPriority());
    }
}
}

class MyThread2 extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(currentThread().getName() + ":" + currentThread().getPriority());
        }
    }
}
}

```

说明：客户线程 t2 的优先级比客户线程 t1 的优先级高，但主线程先启动 t1，可能在主线程一启动 t1，还没有启动 t2 时，主线程就将 CPU 的使用权让给 t1。所以，客户线程 t1 仍然可能先于 t2 得到执行。

程序的一种可能运行结果：

```

Thread-3:10
Thread-3:10
Thread-3:10
Thread-1:5
Thread-3:10
Thread-3:10
Thread-3:10
Thread-3:10
Thread-3:10
Thread-3:10
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5
Thread-1:5

```

11.3.2 线程让步

当前运行的线程调用 `yield()` 方法，使它暂时放弃 CPU，给其他线程一个执行的机会，但当前线程仍处于运行状态，这是因为 Java 中没有进一步区分运行状态与就绪状态。

线程调用 `yield()`，并不一定保证它会放弃 CPU 给其他线程。Java 的线程调度器根据调度算法，仍可能使它继续占用 CPU 运行。若该线程的优先级高，即使它调用 `yield()`，低优先级的线程仍不会占用 CPU 运行。因为 `yield()` 方法只给相同或更高优先级的线程以执行的机会，如果当前系统中没有相同或更高优先级的线程，该方法调用不会产生任何效果，当前线程继续执行。

【例 11.5】 当前线程调用 `yield()` 方法暂时放弃 CPU，给其他线程运行的机会。

YieldThread.java

```

public class YieldThread extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {

```

```

        System.out.println(currentThread().getName() + " ");
        yield();           // 暂时放弃 CPU ， 给其他线程运行的机会
    }
}

public static void main(String[] args) {
    YieldThread m1 = new YieldThread();
    YieldThread m2 = new YieldThread();
    m1.start();
    m2.start();
}
}

```

程序的一种可能运行结果：

```

Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-0
Thread-0

```

`sleep()`方法和 `yield()`方法都是 `Thread()`类的静态方法，都会使当前处于运行状态的线程放弃 CPU，把运行机会让给别的线程。两种的区别在于：

(1) `sleep()`方法会给其他线程运行的机会，而不考虑其他线程的优先级，因此会给较低优先级线程一个运行的机会；`yield()`方法只会给相同优先级或者更高优先级的线程一个运行的机会。

(2) 当线程执行了 `sleep(long millis)`方法后，将转到阻塞状态，参数 `millis` 指定睡眠时间；当线程执行了 `yield()`方法后，线程仍处于就绪状态。

11.3.3 合并线程

一个线程可以在其他线程上调用 `join()`方法，则当前运行的线程将被挂起，直到目标线程运行结束，它才恢复运行。也可以在调用 `join()`方法时带上一个超时参数，这样如果目标线程在这段时间到期时还没有结束的话，`join()`方法总能返回。

【例 11.6】 主线程调用了客户线程的 `join()`方法，主线程将等到客户线程运行结束后，才会恢复运行。

JoinThread.java

```

public class JoinThread {
    public static void main(String[] args) {
        MyThread4 mt = new MyThread4();
        mt.start();
        try {
            mt.join();           // 挂起主线程，等待客户线程执行结束
        } catch (InterruptedException e) {

```

```

    }
    for (int i = 1; i <= 5; i++) {
        System.out.println("main Thread ");
    }
}
}

class MyThread4 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(currentThread().getName());
        }
    }
}
}

```

程序运行结果：

```

Thread-0
Thread-0
Thread-0
Thread-0
Thread-0
main Thread
main Thread
main Thread
main Thread
main Thread

```

11.4 后台线程

后台线程是指为其他线程提供服务的线程，也称为守护线程。Java 虚拟机的垃圾回收线程是典型的后台线程，它负责回收其他线程不再使用的内存。主线程在默认的情况下是前台线程，由前台线程创建的线程在默认的情况下也是前台线程。可通过调用 `Thread` 类的 `setDaemon(true)` 方法，把一个线程设置为后台线程。`Thread` 类的 `isDaemon()` 方法用来判断一个线程是否是后台线程。

【例 11.7】 将 daemon 线程设为后台线程，主线程 `main()` 方法中的代码执行完毕时，由于只剩下后台线程，无论 daemon 是否运行结束，`main()` 主线程都结束，这意味着整个程序运行结束。

SimpleDaemons.java

```

public class SimpleDaemons implements Runnable {
    int i = 0;
    public void run() {
        try {
            while (i < 10) {
                i++;
                System.out.println(Thread.currentThread());
            }
        }
    }
}

```

```

        } catch (Exception e) {
            System.out.println("sleep() interrupted");
        }
    }

    public static void main(String[] args) throws Exception {
        Thread daemon = new Thread(new SimpleDaemons());
        daemon.setDaemon(true);           // 将 daemon 线程设置为后台线程
        daemon.start();                   // 启动后台线程
        System.out.println("main end");
    }
}

```

程序运行结果：

```

main end
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]
Thread[Thread-0,5,main]

```

说明 :若线程 `daemon` 没有被设置为后台线程,则当 `main()`方法中代码执行完毕时, `mian()` 不会结束。它会一直等待 `deamon` 线程运行结束后, `main()`才结束,从而整个程序运行结束。

11.5 综合实例

下面这个实例综合创建多线程,并在程序中使用多线程编程中的常用方法。

【例 11.8】 综合运用多线程的常用方法。使用创建线程的两种方法各创建一个线程。通过继承 `java.lang.Thread` 类来创建 `thread1` 线程,通过实现 `Runnable` 接口来创建 `thread1` 线程。

```

public class MultiThread {
    public static void main(String args[]) {
        System.out.println("我是主线程!");
        SonThread thread1 = new SonThread();           // 创建线程实例 thread1
        // 通过实现 Runnable 接口来创建线程
        Thread thread2 = new Thread(new ThreadRunnable(), "SecondThread");
        thread1.start();                                // 启动线程 thread1 使之处于就绪状态
        thread1.setPriority(7);                          // 设置 thread1 的优先级为 7
        System.out.println("主线程将挂起 10 秒!");
        try {
            Thread.sleep(10000);                        // 主线程挂起 10s
        } catch (InterruptedException e) {
            return;
        }
    }
}

```

```

System.out.println("又回到了主线程!");
if (thread1.isAlive()) {                                // 判断 thread1 是否是活动状态
    thread1.stop();                                     // 如果 thread1 还存在则杀死该线程
    System.out.println("thread1 休眠过长,主线程杀掉了 thread1!");
} else
    System.out.println("主线程未发现 thread1,thread1 已醒顺序执行结束了!");
thread2.start();                                         // 启动 thread2
System.out.println("主线程又将挂起 10 秒!");
try {
    Thread.sleep(10000);                                // 主线程挂起 10s
} catch (InterruptedException e) {
    return;
}
System.out.println("又回到了主线程!");
if (thread2.isAlive()) {                                // 如果 thread2 还存在则杀死该线程
    thread2.stop();
    System.out.println("主线程杀掉了 thread2!");
} else
    System.out.println("thread2 已执行结束了!");
System.out.println("主线程执行结束!");
}
}

class SonThread extends Thread {
    /* 通过继承 Thread 类, 并实现它的抽象方法 run()
    适当时候创建这一 Thread 子类的实例来实现多线程机制*/
    SonThread(){}
    public void run() {
        System.out.println("我是 Thread 子类的线程实例 thread1!");
        System.out.println("我将挂起 10 秒!");
        try {
            sleep(10000);                                // 挂起 10s
        } catch (InterruptedException e) {
            return;
        }
    }
}

class ThreadRunnable implements Runnable {
    /* 通过实现 Runnable 接口中的 run()方法, 再以这个实现了 run()方法的类
    为参数创建 Thread 的线程实例*/
    ThreadRunnable() {}
    public void run() {
        System.out.println("我是 Thread 类的线程实例 thread2 并实现了 Runnable 接口");
        System.out.println("我将挂起 5 秒!");
        try {
            Thread.sleep(5000);                            // 挂起 5s
        } catch (InterruptedException e) {
            return;
        }
    }
}

```



```
    }  
    }  
}
```

程序的一种可能运行结果:

```
我是主线程!  
主线程将挂起10秒!  
我是Thread子类的线程实例thread1!  
我将挂起10秒!  
又回到了主线程!  
主线程未发现thread1,thread1已醒顺序执行结束了!  
主线程又将挂起10秒!  
我是Thread类的线程实例thread2并实现了Runnable接口  
我将挂起5秒!  
又回到了主线程!  
thread2已执行结束了!  
主线程执行结束!
```

第 12 章 综合实例

本章以 Hanoi（汉诺）塔问题为例，综合运用前面各章的知识，用图形用户界面实现汉诺塔的搬运。汉诺塔问题是这样的：古代有一个梵塔，塔内有 3 个座 A、B、C，开始时 A 座上有 64 个盘子，盘子大小不等，大的在下，小的在上。有一个老和尚想把这 64 个盘子从 A 座移到 C 座，但每次只能移动一个盘子，且在移动过程中在 3 个座上都始终保持大盘在下，小盘在上。在移动过程中可以利用 B 座。这里为简单处理，A 座上只放 5 个盘子，5 个盘子既可以移到 C 座，也可以移到 B 座。

12.1 设计思路

设计思路：盘子用 Button 按钮实现，5 个 Button 创建 5 个盘子，用面板 JPanel 来实现梵塔。窗口的上方是一个 Button 按钮，单击该 Button 按钮，程序自动搬运盘子。自动完成盘子的搬运用递归的方式来实现。窗口的右边是信息条，将在信息条上输出用递归方式搬运盘子的整个过程。窗口的下方也是一个 Button 按钮，单击该 Button 按钮，用来重新开始搬运盘子。可以手工把盘子从 A 座搬到 B 座或 C 座上。手工搬运盘子实际上涉及 3 个处理动作：单击 Button 按钮时的处理动作；拖动 Button 按钮时的处理动作；释放 Button 按钮时的处理动作。先粗略看一下这个窗体布局，如图 12.1 所示。

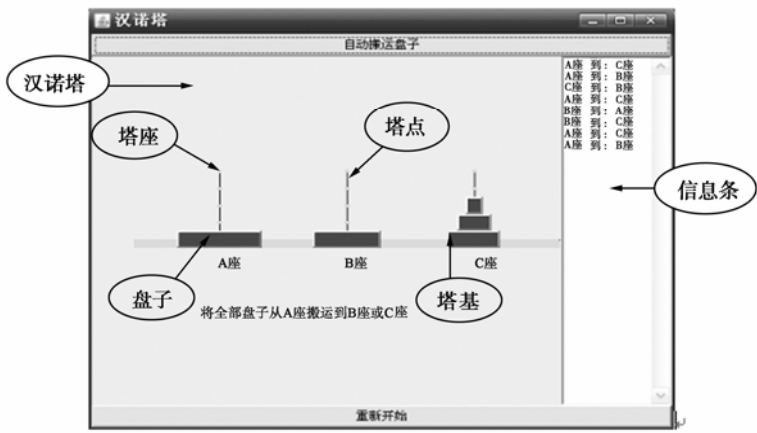


图 12.1 汉诺塔问题

12.2 汉诺塔上盘子模拟

汉诺塔上的盘子用 Button 按钮来模拟。实现手工搬运盘子就是要完成单击盘子，搬运盘

子，存放盘子到目标塔座上。实际上分别对应单击 **Button** 按钮时的处理动作，拖动 **Button** 按钮时的处理动作，释放 **Button** 按钮时的处理动作。**Button** 按钮要对外界的这 3 个鼠标事件做出响应并加以处理，就要在 **Button** 按钮上注册鼠标事件监听器和鼠标移动事件监听器。鼠标事件监听器用于对鼠标单击 **Button** 按钮和释放 **Button** 按钮的动作做出响应和处理。鼠标移动事件监听器用于对鼠标拖动 **Button** 按钮的动作做出响应和处理。

Disk.java

```
package org.hanoiTower;
import java.awt.*;
public class Disk extends Button {
    private int number; // 盘子序号
    private boolean topHaving = false;
    public Disk(int number, HanoiTower con) {
        this.number = number;
        setBackground(Color.blue); // 设置盘子的背景色
        addMouseMotionListener(con); // 注册鼠标移动事件监听器
        addMouseListener(con); // 注册鼠标事件监听器
    }
    public boolean isTopHaving() { // 判断上方是否有盘子
        return topHaving;
    }
    public void setTopHaving(boolean b) { // 设置上方有盘子
        topHaving = b;
    }
    public int getNumber() { // 取得盘子的序号
        return number;
    }
}
```

12.3 汉诺塔上对象的定位及盘子的存放

汉诺塔上包括塔基、塔座、塔点、盘子这一系列对象在整个汉诺塔上的位置都需要定位。对象的定位要由对象的横坐标和纵坐标来确定。盘子的存放由 **deposit()**方法来实现。

TowerPoint.java

```
package org.hanoiTower;
import java.awt.*;
public class TowerPoint {
    private int x, y; // 位置坐标
    private boolean having; // 是否有盘子
    private Disk disk = null;
    private HanoiTower tower = null; // 汉诺塔
    public TowerPoint(int x, int y, boolean having) {
        this.x = x;
        this.y = y;
        this.having = having;
    }
}
```

```

public boolean ishaving() {                                // 判断是否有盘子
    return having;
}
public void setHaving(boolean having) {                    // 设置上方是否有盘子
    this.having = having;
}
public int getX() {
    return x;
}
public int getY() {
    return y;
}
public void deposit(Disk disk, HanoiTower tower) {        // 放置盘子
    this.tower = tower;
    tower.setLayout(null);
    this.disk = disk;
    tower.add(disk);
    int w = disk.getBounds().width;
    int h = disk.getBounds().height;
    disk.setBounds(x - w / 2, y - h / 2, w, h);           // 在指定位置画盘子
    having = true;
    tower.validate();                                     // 画面刷新
}
public Disk getDisk() {                                    // 获取盘子
    return disk;
}
}

```

12.4 创建汉诺塔及实现手动搬运盘子

汉诺塔的创建实际上就是在 **JPanel** 组件上创建塔座、塔基、塔点和需要搬运的盘子。塔座用直线来模拟，为了定位盘子的存放位置，还要在每条直线上画出 5 个椭圆来模拟塔点。3 个塔座的塔基用矩形来实现。手动搬运盘子的 3 个动作就是单击所要搬运的盘子，搬运该盘子，把盘子存放到目标塔座上。因为搬运盘子实际上就是移动 **Button** 按钮，那么手动搬运盘子对应于移动 **Button** 按钮来说的处理动作实际上就是鼠标单击 **Button** 按钮，拖动 **Button** 按钮，释放 **Button** 按钮的处理动作。为了响应和处理鼠标在 **Button** 按钮上的动作，就要在 **Button** 按钮上注册相应的事件监听器，包括鼠标事件监听器和鼠标移动事件监听器。鼠标事件监听器用于处理鼠标单击和释放 **Button** 按钮的事件，鼠标移动事件监听器用于处理鼠标拖动 **Button** 按钮的事件。

HanoiTower.java

```

package org.hanoiTower;
import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.*;
public class HanoiTower extends JPanel implements MouseListener, MouseMotionListener {

```

```

TowerPoint point[];
int x, y;
boolean move = false;
Disk disk[]; // 盘子
int startX, startY; // 盘子所处的位置
int startI; // 单击盘子的序号
int count = 0;
int width, height;
char townername[] = { 'A', 'B', 'C' };
TextArea textArea = null; // 信息条
public HanoiTower(int number, int w, int h, char[] name, TextArea text) {
    townername = name; // 塔座的名字
    count = number; // 盘子的数目
    width = w; // 盘子的宽度
    height = h; // 盘子的高度
    textArea = text;
    setLayout(null); // 取消布局管理器
    addMouseListener(this); // 注册鼠标事件监听器
    addMouseMotionListener(this); // 注册鼠标移动事件监听器
    disk = new Disk[count];
    point = new TowerPoint[3 * count];
    int space = 20;
    for (int i = 0; i < count; i++) {
        point[i] = new TowerPoint(40 + width, 100 + space, false); // 创建左边的塔点
        space = space + height;
    }
    space = 20;
    for (int i = count; i < 2 * count; i++) {
        point[i] = new TowerPoint(160 + width, 100 + space, false); // 创建中间的塔点
        space = space + height;
    }
    space = 20;
    for (int i = 2 * count; i < 3 * count; i++) {
        point[i] = new TowerPoint(280 + width, 100 + space, false); // 创建右边的塔点
        space = space + height;
    }
    int tempWidth = width;
    int sub = (int) (tempWidth * 0.2);
    for (int i = count - 1; i >= 0; i--) {
        disk[i] = new Disk(i, this); // 创建盘子
        disk[i].setSize(tempWidth, height); // 设置盘子的大小
        tempWidth = tempWidth - sub;
    }
    for (int i = 0; i < count; i++) {
        point[i].deposit(disk[i], this); // 在左边的塔点上放置盘子
        if (i >= 1)
            disk[i].setTopHaving(true); // 设置上方有盘为 true
    }
}

```

```

    }
}
//*****创建汉诺塔*****
public void paintComponent(Graphics g) {
    ...
}
//*****单击盘子的处理动作*****
public void mousePressed(MouseEvent e) {
    ...
}
public void mouseMoved(MouseEvent e) {
}
public void mouseEntered(MouseEvent e) {
}
//*****搬运盘子的处理动作*****
public void mouseDragged(MouseEvent e) {
    ...
}
//*****存放盘子的处理动作*****
public void mouseReleased(MouseEvent e) {
    ...
}
public void mouseExited(MouseEvent e) {
}
public void mouseClicked(MouseEvent e) {
}
}
}

```

其中：

● 创建汉诺塔

```

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    //第一个塔座
    g.drawLine(point[0].getX(), point[0].getY(), point[count - 1].getX(), point[count - 1].getY());
    g.drawLine(point[count].getX(), point[count].getY(), point[2 * count - 1]
        .getX(), point[2 * count - 1].getY()); // 第二个塔座
    g.drawLine(point[2 * count].getX(), point[2 * count].getY(),
        point[3 * count - 1].getX(), point[3 * count - 1].getY()); // 第三个塔座
    g.drawLine(point[count - 1].getX() - width, point[count - 1].getY(),
        point[3 * count - 1].getX() + width, point[3 * count - 1].getY());
    int leftx = point[count - 1].getX() - width; // 获取塔基的横坐标
    int lefty = point[count - 1].getY(); // 获取塔基的纵坐标
    int w = (point[3 * count - 1].getX() + width) - (point[count - 1].getX() - width); // 获取塔基的宽度
    int h = height / 2; // 获取汉诺塔的高度
    g.setColor(Color.green); // 设置汉诺塔的颜色
    g.fillRect(leftx, lefty, w, h); // 创建塔基
    int size = 4;
}

```

```

for (int i = 0; i < 3 * count; i++) { // 创建塔点
    g.fillOval(point[i].getX()- size / 2, point[i].getY()- size / 2,size, size);
}
g.drawString("" + townername[0] + "塔", point[count - 1].getX(),point[count - 1].getY() + 30);
g.drawString("" + townername[1] + "塔", point[2 * count - 1].getX(),point[count - 1].getY() + 30);
g.drawString("" + townername[2] + "塔", point[3 * count - 1].getX(),point[count - 1].getY() + 30);
g.drawString("将全部盘子从" + townername[0] + "塔搬运到" + townername[1] + "塔或"
    + townername[2] + "塔", point[count - 1].getX(), point[count - 1].getY() + 80);
}

```

● 单击盘子时的处理动作

```

public void mousePressed(MouseEvent e) {
    Disk disk = null;
    Rectangle rect = null;
    if (e.getSource() == this)
        move = false;
    if (move == false){
        if (e.getSource() instanceof Disk) { // 判断当前的对象是否是盘子
            disk = (Disk) e.getSource();
            startX = disk.getBounds().x; // 获取盘子的横坐标
            startY = disk.getBounds().y; // 获取盘子的纵坐标
            rect = disk.getBounds();
            for (int i = 0; i < 3 * count; i++) {
                int x = point[i].getX(); // 获取塔点的横坐标
                int y = point[i].getY(); // 获取塔点的纵坐标
                if (rect.contains(x, y)) {
                    startI = i; // 获取鼠标单击的是哪个盘子
                    break;
                }
            }
        }
    }
}

```

● 搬运盘子时的处理动作

```

public void mouseDragged(MouseEvent e) {
    Disk disk = null;
    if (e.getSource() instanceof Disk) {
        disk = (Disk) e.getSource();
        move = true;
        e = SwingUtilities.convertMouseEvent(disk, e, this); // 将鼠标事件从盘子转到当前汉诺塔上
    }
    if (e.getSource() == this) {
        if (move && disk != null) {
            x = e.getX();
            y = e.getY();
            if (disk.isTopHaving() == false)
                disk.setLocation(x - disk.getWidth() / 2, y - disk.getHeight() / 2);
        }
    }
}

```

```

    }
}
}

```

● 存放盘子时的处理动作

```

public void mouseReleased(MouseEvent e) {
    Disk disk = null;
    move = false;
    Rectangle rect = null;
    if (e.getSource() instanceof Disk) {
        disk = (Disk) e.getSource();
        rect = disk.getBounds();
        e = SwingUtilities.convertMouseEvent(disk, e, this);
    }
    if (e.getSource() == this) {
        boolean containTowerPoint = false;
        int x = 0, y = 0;
        int endI = 0;
        if (disk != null) {
            for (int i = 0; i < 3 * count; i++) {
                x = point[i].getX();
                y = point[i].getY();
                if (rect.contains(x, y)) {
                    containTowerPoint = true;
                    endI = i;
                    break;
                }
            }
        }
        if (disk != null && containTowerPoint) {
            if (point[endI].ishaving() == true) {
                // 盘子没拿走
                disk.setLocation(startX, startY);
                // 放回原地
            } else {
                //如果是塔座的最低点，可直接放上盘子
                if (endI == count - 1 || endI == 2 * count - 1 || endI == 3 * count - 1) {
                    point[endI].deposit(disk, this);
                    if (startI != count - 1 && startI != 2 * count - 1 && startI != 3 * count - 1) {
                        (point[startI + 1].getDisk()).setTopHaving(false); // 盘子已经被取走
                        point[startI].setHaving(false);
                    } else {
                        point[startI].setHaving(false);
                    }
                } else {
                    // 判断下面是否有盘子
                    if (point[endI + 1].ishaving() == true) {
                        Disk tempDisk = point[endI + 1].getDisk();
                        // 如果上面的盘子比下面的盘子小，可以放上盘子

```



```

        if ((tempDisk.getNumber() - disk.getNumber()) >= 1) {
            point[endI].deposit(disk, this);
            if (startI != count - 1 && startI != 2 * count - 1
                && startI != 3 * count - 1) {
                // 盘子已经被取走
                (point[startI + 1].getDisk()).setTopHaving(false);
                point[startI].setHaving(false);           // 盘子已经被取走
                // endI+1 位置的上方 endI 的盘子就是刚刚放上的盘子
                tempDisk.setTopHaving(true);
            } else {
                point[startI].setHaving(false);           // 该方向的盘子已经
                                                         被取完
                tempDisk.setTopHaving(true);
            }
        } else {
            disk.setLocation(startX, startY);
        }
    } else {
        disk.setLocation(startX, startY);
    }
}

}

if (disk != null && !containTowerPoint) {
    disk.setLocation(startX, startY);
}

}

}

```

HanoiTowerCarry.java 是主程序，布局整个窗体。

HanoiTowerCarry.java

```

package org.hanoiTower;
import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.*;
public class HanoiTowerCarry extends Frame implements ActionListener {
    HanoiTower tower = null;
    Button renew = null;
    char townername[] = { 'A', 'B', 'C' };
    int count, width, height;
    Thread thread;                                     // 线程
    TextArea textArea = null;                          // 信息条
    public HanoiTowerCarry () {
        count = 5;
        width = 80;
        height = 18;
        textArea = new TextArea(12, 12);
    }
}

```

```

        textArea.setText(null);
        tower = new HanoiTower(count, width, height, tovername, textArea);
        renew = new Button("重新开始");
        renew.setBackground(Color.cyan);
        renew.addActionListener(this);
        add(tower, BorderLayout.CENTER);
        add(renew, BorderLayout.SOUTH);
        add(textArea, BorderLayout.EAST);
        addWindowListener(new WindowAdapter() {           // 注册监听器
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setVisible(true);
        setBounds(60, 20, 670, 540);
        validate();                                       // 重新布局窗体
    }

    public void actionPerformed(ActionEvent e) {         // 判断单击的是哪个按钮
        if (e.getSource() == renew) {                   // 单击的是“重新开始”按钮
            this.remove(tower);
            textArea.setText(null);
            tower = new HanoiTower(count, width, height, tovername, textArea);
            add(tower, BorderLayout.CENTER);
            validate();
        }
    }

    public static void main(String args[]) {
        new HanoiTowerCarry ().setTitle("汉诺塔");
    }
}

```

运行 HanoiTowerCarry.java 程序。现在可以手动把 5 个盘子符合规则地搬到 B 座或 C 座，界面如图 12.2 所示。

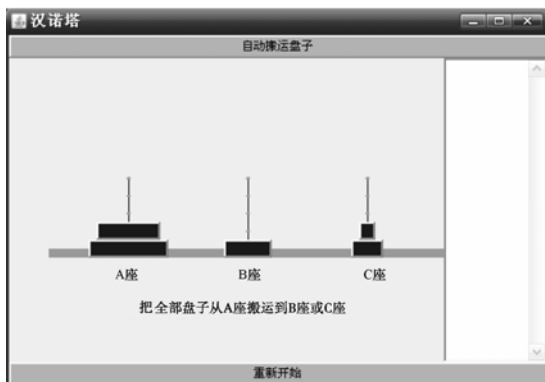


图 12.2 手动搬运盘子

12.5 自动搬运盘子

既然可以实现用手工把盘子搬到目标塔座上的功能，那么同样可以实现自动搬运盘子的功能。自动搬运盘子的功能可用递归方式来实现。

那么设计递归的思路是怎样的呢？如果 A 塔上只有一个盘子，这个很容易，直接把 A 塔上的盘子搬到 C 塔上即可。但是现在 A 塔上有 5 个盘子，那么可考虑如下处理。

- (1) 将 A 塔上的 4 个盘子借助 C 塔先移到 B 塔上。
- (2) 把 A 塔上剩下的 1 个盘子移到 C 塔上。
- (3) 将 4 个盘子从 B 塔借助 A 塔移到 C 塔上。

为了实现上述目标，设计的递归算法如下。

```
void Hanoi( int n, char A, char B,char C) {
    if ( n == 1)
        move(A,C);           // 只有一个盘子，直接从 A 座搬到 C 座上
    else {
        hanoi (n-1,A,C,B);    // 将 A 座上的 n-1 个盘子借助 C 座先移到 B 座上
        move(A,C);
        hanoi (n-1,B,A,C);    // 将 n-1 个盘子从 B 座借助 A 座移到 C 座上
    }
}
```

实现上述递归算法的程序，也就是实现这些功能：判断塔座上的盘子数，如果是一个盘子，直接把盘子搬到目标塔座上；如果不是一个盘子，则要使用递归方法分步搬运。但无论是一个盘子还是多个盘子，搬运盘子的实际操作需要后面的功能来完成：获得在源塔座中最上面的盘子；获取盘子在该塔座中的位置；获取目标塔座的最上方盘子的上方位置。为什么需要后面的两项功能呢？因为汉诺塔的搬运有一个要求，那就是大的盘子在下，小的盘子在上。如果源塔座上的盘子比目标塔座上的盘子要大，那么则不能搬运到目标塔座上。

实现此递归算法的程序 HanoiTower.java 如下所示。同 12.4 节相同的 HanoiTower.java 程序将不在此列出。

HanoiTower.java

```
package org.hanoiTower;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class HanoiTower extends JPanel implements MouseListener,MouseMotionListener {
    /******* 自动搬运盘子*****
    public void autoCarry(int count, char A, char B, char C) {
        if(count == 1) {
            textArea.append("'" + A + " 到: " + C + "座\n");
            Disk disk = getTopDisk(A);           // 在塔中获取最上面的盘子
            int startI = getTopPosition(A);       // 在塔中获取最上面盘子的位置
            int endI = getSupremePosition(C);     // 在塔中获取最上面盘子的上方位置
            if (disk != null) {
                point[endI].deposit(disk, this);
            }
        }
    }
}
```

```

        point[startI].setHaving(false);
    try {
        Thread.sleep(1000);
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}
} else {
    autoCarry(count - 1, A, C, B);           // 将 A 座上的 count-1 个盘子借助 C 座先移到
                                            // B 座上
    textArea.append("" + A + " 到: " + C + "塔\n");
    Disk disk = getTopDisk(A);               // 在塔中获取最上面的盘子
    int startI = getTopPosition(A);           // 在塔中获取最上面盘子的位置
    int endI = getSupremePosition(C);         // 在塔中获取最上面盘子的上方位置
    if (disk != null) {
        point[endI].deposit(disk, this);
        point[startI].setHaving(false);
        try {
            Thread.sleep(1000);
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
    autoCarry(count - 1, B, A, C);           // 将 count-1 个盘子从 B 座借助 A 座移到 C 座上
}
}
//*****获得源塔座中最上方的盘子*****
public Disk getTopDisk(char name) {
    Disk disk = null;
    if (name == tovername[0]) {              // 第一个塔座
        for (int i = 0; i < count; i++) {
            if (point[i].ishaving() == true) { // 判断是否有盘子
                disk = point[i].getDisk();     // 获取盘子
                break;
            }
        }
    }
    if (name == tovername[1]) {              // 第二个塔座
        for (int i = count; i < 2 * count; i++) {
            if (point[i].ishaving() == true) {
                disk = point[i].getDisk();
                break;
            }
        }
    }
    if (name == tovername[2]) {              // 第三个塔座
        for (int i = 2 * count; i < 3 * count; i++) {

```

```

        if (point[i].ishaving() == true) {
            disk = point[i].getDisk();
            break;
        }
    }
}
return disk;
}

//*****获取盘子在该塔座中的位置*****
public int getSupremePosition(char name) {
    int position = 0;
    if (name == tovername[0]) {           // 判断塔座的名字
        int i = 0;
        for (i = 0; i < count; i++) {
            if (point[i].ishaving() == true) {    // 判断是否有盘子
                position = Math.max(i - 1, 0);
                break;
            }
        }
        if (i == count) {
            position = count - 1;
        }
    }
    if (name == tovername[1]) {
        int i = 0;
        for (i = count; i < 2 * count; i++) {
            if (point[i].ishaving() == true) {
                position = Math.max(i - 1, 0);
                break;
            }
        }
        if (i == 2 * count) {
            position = 2 * count - 1;
        }
    }
    if (name == tovername[2]) {
        int i = 0;
        for (i = 2 * count; i < 3 * count; i++) {
            if (point[i].ishaving() == true) {
                position = Math.max(i - 1, 0);
                break;
            }
        }
        if (i == 3 * count) {
            position = 3 * count - 1;
        }
    }
}

```

```

        return position;
    }
    /*******获取目标塔座中最上方盘子的上方位置*****
    public int getTopPosition(char name) {
        int position = 0;
        if (name == tovername[0]) {           // 判断塔座的名字
            int i = 0;
            for (i = 0; i < count; i++) {
                if (point[i].ishaving() == true) {    // 判断是否有盘子
                    position = i;
                    break;
                }
            }
            if (i == count) {
                position = count - 1;
            }
        }
        if (name == tovername[1]) {
            int i = 0;
            for (i = count; i < 2 * count; i++) {
                if (point[i].ishaving() == true) {
                    position = i;
                    break;
                }
            }
            if (i == 2 * count) {
                position = 2 * count - 1;
            }
        }
        if (name == tovername[2]) {
            int i = 0;
            for (i = 2 * count; i < 3 * count; i++) {
                if (point[i].ishaving() == true) {
                    position = i;
                    break;
                }
            }
            if (i == 3 * count) {
                position = 3 * count - 1;
            }
        }
        return position;
    }
}

```

HanoiTowerCarry.java 是主程序，创建窗体等其他按钮组件。

HanoiTowerCarry.java

```

package org.hanoiTower;
import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.*;
public class HanoiTowerCarry extends Frame implements ActionListener, Runnable {
    HanoiTower tower = null;
    Button renew, auto = null;
    char tovername[] = { 'A', 'B', 'C' };
    int count, width, height;
    Thread thread; // 线程
    TextArea textArea = null; // 信息条
    public HanoiTowerCarry () {
        thread = new Thread(this);
        count = 5;
        width = 80;
        height = 18;
        textArea = new TextArea(12, 12);
        textArea.setText(null);
        tower = new HanoiTower(count, width, height, tovername, textArea);
        renew = new Button("重新开始");
        renew.setBackground(Color.cyan);
        auto = new Button("自动搬运盘子");
        auto.setBackground(Color.cyan);
        renew.addActionListener(this);
        auto.addActionListener(this);
        add(tower, BorderLayout.CENTER);
        add(renew, BorderLayout.SOUTH);
        add(auto, BorderLayout.NORTH);
        add(textArea, BorderLayout.EAST);
        addWindowListener(new WindowAdapter() { // 注册监听器
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setVisible(true);
        setBounds(60, 20, 670, 540);
        validate(); // 重新布局窗体
    }
    public void actionPerformed(ActionEvent e) { // 判断单击的是哪个按钮
        if (e.getSource() == renew) { // 重新开始
            if (!(thread.isAlive())) {
                this.remove(tower);
                textArea.setText(null);
                tower = new HanoiTower(count, width, height, tovername, textArea);
                add(tower, BorderLayout.CENTER);
            }
        }
    }
}

```

```

        validate();
    }
}
if (e.getSource() == auto) { // 自动搬运盘子
    if (!(thread.isAlive())) {
        thread = new Thread(this);
    }
    try {
        thread.start(); // 启动线程
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}
}
public void run() {
    this.remove(tower);
    textArea.setText(null);
    tower = new HanoiTower(count, width, height, tovername, textArea);
    add(tower, BorderLayout.CENTER);
    validate();
    tower.autoCarry(count, tovername[0], tovername[1], tovername[2]);
}
public static void main(String args[]) {
    new HanoiTowerCarry ().setTitle("汉诺塔");
}
}

```

运行 HanoiTowerCarry.java 程序，界面如图 12.3 所示。

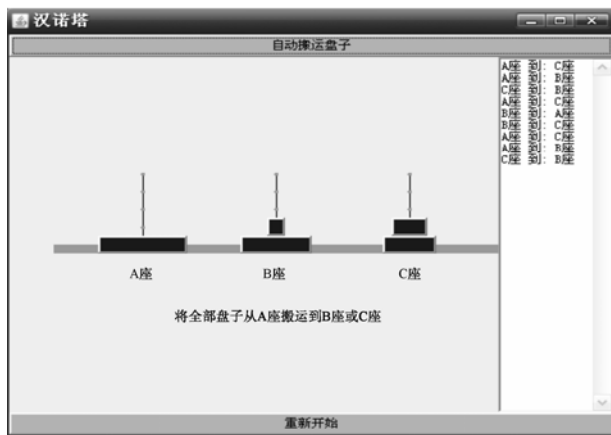


图 12.3 自动搬运盘子

第 13 章 Java 网络编程

网络应用程序就是在已实现网络互联的不同计算机上运行的应用程序，这些程序之间可以相互交换数据。Java 是优秀的网络编程语言，Java 网络编程的类库位于 `java.net` 包中。Java 支持 TCP/UDP 及其上层的网络编程，对 TCP/UDP 以下层，如 IP 包的捕获、侦听、数据链路层帧的捕获，需要借助第三方的 java 包，如 UNIX/Linux 下著名的 libpcap 包的 Java 版本 jpcap 包。

这里主要讲述基于 `java.net` 包中的类进行网络编程技术，分为 3 个部分：基于 TCP/IP 协议基础上的 TCP 通信、UDP 通信和 URL 通信。

13.1 网络程序设计基础

在计算机网络中，主机与主机之间互相通信，就必须让通信双方知道对方的地址。就像邮递信件一样，发信人必须在信件上写明收信人的详细地址信息，这样，邮递员才能根据这个地址信息准确投递到收信人的信箱里。同样，在计算机网络中，发送方必须知道接收方主机的地址信息。也就是说，必须为每台计算机指定一个地址。采取的方法就是使用 IP 地址来标识网络中的每一台主机。

13.1.1 TCP和UDP

要想让处于网络中的主机互相通信，只是知道通信双方地址还是不够的，还必须遵循一定的规则。有两套参考模型：OSI 参考模型、TCP/IP 参考模型（或 TCP/IP 协议）。由于 OSI 参考模型过于理想化，未能在因特网上进行广泛推广。这样，TCP/IP 协议就成为事实上的国际标准。这里只重点介绍 TCP/IP 协议的运输层协议，运输层协议中有两个非常重要的协议：传输控制协议 TCP（Transmission Control Protocol）、用户数据报协议 UDP（User Datagram Protocol）。

传输控制协议 TCP 是面向连接的运输层协议，即应用进程（或程序）在使用 TCP 协议之前，必须先建立 TCP 连接，在传输完毕后，释放已经建立的连接。利用 TCP 协议进行通信的两个应用进程，一个是服务器进程，另一个是客户进程。

用户数据报协议 UDP 是面向无连接的运输层协议，即应用进程（或程序）在使用 UDP 协议之前，不必先建立连接。自然，发送数据结束时也没有连接需要释放。因此，减少了开销和发送数据之前的时延。

13.1.2 端口和套接字

一般情况下，两台主机上都会运行许多进程。当主机 A 上的进程 A1 向主机 B 上的进程 B1 发送数据时，IP 协议根据主机 B 的 IP 地址，把进程 A1 发送的数据送达主机 B。接下来运输层 TCP 需要决定把数据发送到主机中的哪个进程。TCP 和 UDP 都采用端口来区分进程。

端口是一种抽象的软件结构（包括一些数据结构和 I/O 缓冲区），也称为协议端口（protocol

port)。端口用一个整数型标识符来表示，即端口号。端口号跟协议相关，TCP/IP 传输层的两个协议 TCP 和 UDP 是完全独立的两个软件模块，因此各自的端口号也相互独立，端口使用一个 16 位的数字来表示，它的范围是 0~65535，1024 以下的端口号保留给预定义的服务，用于一些知名的网络服务和应用。例如，HTTP 使用 80 端口，FTP 服务的端口号是 21。应用程序通过系统调用与某端口建立连接（binding）后，传输层传给该端口的数据都被相应的进程所接收，相应进程发给传输层的数据都通过该端口输出。

在计算机网络的定义中，端口号拼接到 IP 地址即构成套接字（Socket）。在客户/服务器通信模式中，客户端需要主动创建与服务器连接的 Socket，服务器进程收到了客户端的连接请求时，也会创建与客户连接的 Socket。Socket 可看做通信连接两端的收发器，服务器进程与客户进程都通过 Socket 来收发数据。在一个 Socket 对象中同时包含了远程服务器的 IP 地址和端口信息，以及本地客户的 IP 地址和端口信息。从 Socket 对象中还可以获得输入流和输出流，分别用于接收从服务器端发来的数据，以及向服务器发送数据。

13.2 TCP网络编程

两个 Java 应用程序可通过一个双向的网络通信连接实现数据交换，这个双向链路的一段称为一个 Socket（套接字）。Socket 通常用来实现 Client/Server 连接。Socket 的出现，使程序员可以很方便地访问 TCP/IP，从而开发各种网络应用程序。

Java 语言的基于套接字编程分为服务器编程和客户端编程，其通信模型如图 13.1 所示。

● 服务器程序编写

- （1）调用 `ServerSocket(int port)` 创建一个服务器端套接字，并绑定到指定端口上。
- （2）调用 `accept()`，监听连接请求，如果客户端请求连接，则接收连接，返回通信套接字。
- （3）调用 Socket 类的 `getOutputStream` 和 `getInputStream` 获取输出流和输入流，开始网络数据的发送和接收。
- （4）最后关闭通信套接字。

● 客户端程序编写

- （1）调用 `Socket()` 创建一个流套接字，并请求连接到服务器端。
- （2）调用 Socket 类的 `getOutputStream` 和 `getInputStream` 获取输出流和输入流，开始网络数据的发送和接收。
- （3）最后关闭通信套接字。

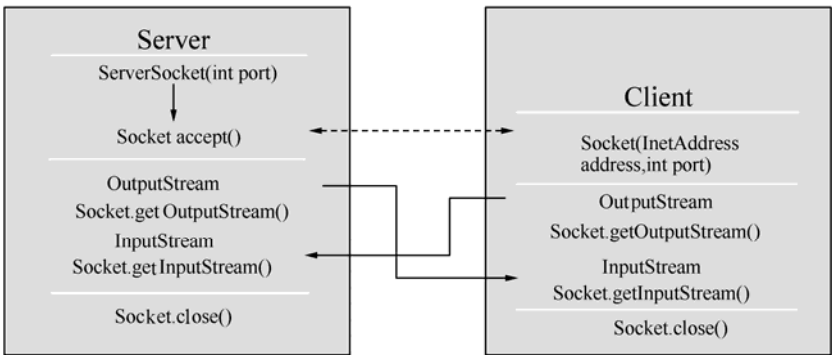


图 13.1 基于 TCP 的 Socket 通信

Socket 类的对象表示一个 Socket。客户端使用 Socket 类的构造方法，创建一个 Socket 对象，创建的同时会自动向服务器方发起连接。

Socket 类的构造方法如下。

● Socket(String host,int port)throws UnknownHostException,IOException

功能：向服务器（域名是 host，端口号为 port）发起 TCP 连接，若成功，则创建 Socket 对象，否则抛出异常。

● Socket(InetAddress address,int port)throws IOException

功能：同上。只是根据 InetAddress 对象所表示的 IP 地址以及端口号 port 发起连接。

● Socket(String host,int port,InetAddress localAddr,int localPort)throws IOException

功能：创建一个套接字并将其连接到指定远程主机上的指定远程端口。Socket 会通过调用 bind()方法来绑定提供的本地地址及端口。host 表示远程主机名，port 表示远程端口号，localAddr 表示要将套接字绑定到的本地地址，localPort 表示要将套接字绑定到的本地端口。

● Socket(InetAddress address,int port,InetAddress localAddr,int localPort) throws IOException

功能：创建一个套接字并将其连接到指定远程地址上的指定远程端口。Socket 会通过调用 bind()方法来绑定提供的本地地址及端口。

Socket 类的常用方法如表 13.1 所示。

表 13.1 Socket 类的常用方法

方 法	功 能
InetAddress getLocalAddress()	返回对方 Socket 中的 IP 的 InetAddress 对象
int getLocalPort()	返回本地 Socket 中的端口号
InetAddress getInetAddress()	返回对方 Socket 中的 IP 地址
int getPort()	返回对方 Socket 中的端口号
Void close() throws IOException	关闭 Socket，释放资源
InputStream getInputStream()throws IOException	获取与 Socket 相关联的字节输入流，用于从 Socket 中读数据
OutputStream getOutputStream()throws IOException	获取与 Socket 相关联的字节输出流，用于向 Socket 中写数据

服务器端需要创建监听特定端口的 ServerSocket，ServerSocket 等待客户端发起 TCP 连接，然后返回一个用于与该客户端进行 TCP 通信的 Socket 对象。

ServerSocket 类的构造方法如下。

● ServerSocket(int port)throws IOException

功能：创建绑定到特定端口的服务器套接字。连接队列的最大长度是 50，当连接队列已满，又有客户端发起连接请求时，服务器将拒绝该连接请求（连接队列是指已完成 TCP 三次握手但还没有被 accept()取走的 TCP 连接）。

● ServerSocket(int port,int backlog)throws IOException

功能：利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。backlog 表示队列的最大长度。

● ServerSocket(int port,int backlog,InetAddress bindAddr)throws IOException

功能：使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器，bindAddr 表示要将服务器绑定到的 InetAddress。

ServerSocket 类的常用方法如表 13.2 所示。

表 13.2 ServerSocket 类的常用方法

方 法	功 能
Socket accept() throws IOException	等待客户端的连接请求，返回与该客户端进行通信的 Socket 对象
void setSoTimeout(int timeout) throws SocketException	设置 accept()方法等待连接的时间为 timeout 毫秒。若时间已到，还没有客户端连接，则抛出 InterruptedIOException 异常，accept()方法不再阻塞，该监听 Socket 可继续使用。若 timeout 值为 0，则表示 accept()永远等待。该方法必须在监听 Socket 创建后，在 accept()之前调用才有效
void close()throws IOException	关闭监听 Socket
InetAddress getInetAddress()	返回此服务器套接字的本地地址
int getLocalPort()	返回此套接字在其上监听的端口号
SocketAddress getLocalSocketAddress()	返回此套接字绑定的端点的地址

13.2.1 InetAddress类

InetAddress 类的对象用于表示 IP 地址，该类没有明确定义构造方法，常用的方法如表 11.3 所示。

表 13.3 InetAddress 类的常用方法

方 法	功 能
static InetAddress[] getAllByName(String host) throws UnknownHostException	返回主机名 host 所对应的所有 IP，每一个 IP 用一个 InetAddress 对象表示，结果返回的是一个一维的 InetAddress 数组
static InetAddress getByName(String host) throws UnknownHostException	返回主机名 host 所对应的一个 IP。若该主机名对应多个 IP，则随机返回其中一个 IP，该 IP 用 InetAddress 对象表示
static InetAddress getLocalHost() throws UnknownHostException	返回本地主机的 IP 地址。该 IP 用 InetAddress 对象表示
public byte[] getAddress()	返回组成该 IP 地址的 4 个字节。按网络字节存放，即最高字节放在 getAddress()[0]中
static InetAddress getByAddress(byte[] addr) throws UnknownHostException	返回由该 4 个字节组成的 IP 地址的 InetAddress 对象
byte[] getAddress()	返回 IP 地址的 4 个字节组成的数组

【例 13.1】 返回域名相应的 IP 地址，若没有给出域名，则返回本地主机的 IP 地址。

InetAddressTest.java

```
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class InetAddressTest {
    static TextField tf1 = new TextField(40);
    static List list = new List(6);
```

```

public static void main(String[] args) throws Exception {
    Frame f = new Frame();
    f.add(list);
    f.setSize(300, 300); // 设置窗体的大小
    Panel p = new Panel();
    p.setLayout(new BorderLayout()); // 设置边界布局管理器
    tf1.addActionListener(new MyListener()); // 注册事件监听器
    p.add("West", tf1);
    f.add("South", p);
    f.addWindowListener(new WindowAdapter() { // 关闭窗口
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    f.setVisible(true);
}

class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String s = InetAddressTest.tf1.getText(); // 获取文本框中的内容
        InetAddress[] addr;
        try {
            InetAddressTest.list.removeAll(); // 将列表框中的原有内容清除
            addr = InetAddress.getAllByName(s); // 返回主机名所对应的所有 IP 地址
            for (int i = 0; i < addr.length; i++) {
                InetAddressTest.list.add(addr[i].toString()); // 添加到列表框中
            }
        } catch (UnknownHostException e1) {
            e1.printStackTrace();
        }
        ((TextField) e.getSource()).setText(null); // 设置文本框的内容为空
    }
}

```

运行程序，在文本框中输入新浪的域名“www.sina.com”，则在列表框中显示新浪的 IP 地址信息，如图 13.2 所示。

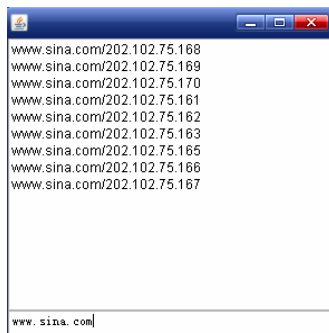


图 13.2 获取新浪的 IP 地址

13.2.2 TCP通信程序

【例 13.2】 一个简单 TCP 通信程序，客户端程序向服务器端程序发送任意的字符串，服务器端程序收到后，显示收到的字符串。

首先建立一个 TCP 服务器端程序。在这个程序中，创建一个在 8866 端口上等待连接的 `ServerSocket` 对象。当接收到一个客户端的连接请求后，程序从与这个客户建立连接的 `Socket` 对象中获得输入流对象。通过输入流读取客户端程序发送的字符串。

TCP 服务器端 `TCPServer.java`

```
package com.net;
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class TCPServer {
    static DataInputStream dis = null;
    public static void main(String[] args) {
        boolean started = false;
        Socket s = null;
        TextArea ta = new TextArea();
        ta.append("从客户端接收的数据: "+"\\n");
        ServerSocket ss = null;
        try {
            ss = new ServerSocket(8866);           // 创建一个监听 Socket 对象
        } catch (BindException e) {
            System.exit(0);
        } catch (IOException e) {
            e.printStackTrace();
        }
        Frame f = new Frame("服务器端");
        f.setLocation(300, 300);
        f.setSize(200, 200);
        f.add(ta, BorderLayout.NORTH);
        f.pack();
        f.addWindowListener(new WindowAdapter() { // 关闭窗口
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        f.setVisible(true);                       // 设置窗体可见
        try {
            started = true;
            while (started) {
                boolean bConnected = false;
                s = ss.accept();                   // 等待客户端请求连接
                bConnected = true;
            }
        }
    }
}
```

```

        dis = new DataInputStream(s.getInputStream());
        while (bConnected) {
            String str = dis.readUTF();           // 从输入流中读取数据
            ta.append(str+"\n");                  // 将数据添加到文本区中
        }
    }
} catch (EOFException e) {
    System.out.println("Client closed!");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (dis != null)
            dis.close();                         // 关闭输入流
        if (s != null)
            s.close();                           // 关闭 Socket 对象
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

再建立 TCP 客户端程序。由于是本地连接，TCP 服务器的 IP 地址是 127.0.0.1，客户端需要与通过服务器的 8866 端口建立连接完成通信，所以在客户端指定要与服务器连接的端口号为 8866。同样，客户端利用创建的 Socket 对象来获得输出流对象。通过输出流向服务器端发送字符串。

TCP 客户端 TCPClient.java

```

package com.net;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class TCPClient extends Frame {
    Socket s = null;
    DataOutputStream dos = null;
    DataInputStream dis = null;
    TextField tf = new TextField(40);
    List list = new List(6);
    public static void main(String[] args) {
        TCPClient client = new TCPClient();
        client.list.add("向服务器端发送的数据: ");
        client.setTitle("客户端");
        client.run();
    }
    public void run() {
        setLocation(400, 300);                  // 设置窗体的位置
    }
}

```

```

this.setSize(300, 300); // 设置窗体的大小
add(tf, BorderLayout.SOUTH);
add(list, BorderLayout.NORTH);
pack();
this.addWindowListener(new WindowAdapter() { // 关闭窗口体
    public void windowClosing(WindowEvent e) {
        disconnect();
        System.exit(0);
    }
});
tf.addActionListener(new MyListener()); // 注册事件监听器
setVisible(true);
connect();
}
public void connect() {
    try {
        s = new Socket("127.0.0.1", 8866); // 创建一个向服务器端发起连接的 Socket 对象
        dos = new DataOutputStream(s.getOutputStream());
    } catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
public void disconnect() {
    try {
        dos.close(); // 关闭输出流
        s.close(); // 关闭 Socket 对象
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
private class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String s1 = null;
        String s2 = null;
        String str = tf.getText().trim(); // 获取文本框中的内容
        list.add(str);
        tf.setText(""); // 将文本框的内容清空
        try {
            dos.writeUTF(str); // 向输出流中写入数据
            dos.flush(); // 刷新输出流
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
}
}

```


}

运行上面的 TCP 客户端服务器程序。首先在客户端文本框中输入字符串，按下回车键，这些字符串将显示在客户端的窗口上。并且，这些数据将发送给服务器端，服务器端接收到从客户端发送的字符串，将显示在窗口上，如图 13.3 、图 13.4 所示。



图 13.3 服务器端



图 13.4 客户端

13.3 UDP网络编程

采用 TCP 协议通信时，客户端的 Socket 必须先与服务器端建立连接，连接建立成功后，服务器端也会持有与客户端连接的 Socket。客户端的 Socket 与服务器端的 Socket 是对应的，它们构成了两个端点之间的虚拟通信链路。与 TCP 通信不同，UDP 是面向无连接、不可靠的基于数据包的传输协议。

在 Java 中，`java.net.DatagramSocket` 负责接收和发送 UDP 数据报，`java.net.DatagramPacket` 表示 UDP 数据报。每个 `DatagramSocket` 与一个数据报套接字（包括本地主机的 IP 地址和本地 UDP 端口）绑定，每个 `DatagramSocket` 可以把 UDP 数据报发送给任意一个远程 `DatagramSocket`，也可以接收来自任意一个远程 `DatagramSocket` 的数据报。在 UDP 数据报中包含了目的地址信息，`DatagramSocket` 可以根据该信息把数据报发送到目的地。

UDP 协议是无连接的协议。客户端的 `DatagramSocket` 与服务器端的 `DatagramSocket` 不存在一一对应关系，两者无须建立连接，就能交换数据报。每个 `DatagramSocket` 对象都会与一个本地端口绑定，在此端口监听发送过来的数据报。在服务器程序中，一般由程序显式地为 `DatagramSocket` 指定本地端口。在客户程序中，一般由操作系统为 `DatagramSocket` 分配本地端口，这种端口也称为匿名端口。

● `DatagramSocket` 的构造方法如下。

`DatagramSocket()`throws `SocketException`

功能：构造数据报套接字并将其绑定到本地主机上任何可用的端口。套接字将被绑定到 `INADDR_ANY` 地址，IP 地址由内核来选择。

`DatagramSocket(int port)`throws `SocketException`

功能：创建数据报套接字并将其绑定到本地主机上的指定端口。套接字将被绑定到 `INADDR_ANY` 地址，IP 地址由内核来选择。

`DatagramSocket` 类的常用方法如表 13.4 所示。

表 13.4 DatagramSocket 类的常用方法

方 法	功 能
void send(DatagramPacket p) throws IOException	发送一个 UDP 数据包。一个 UDP 数据包就是一个 DatagramPacket 对象
void receive(DatagramPacket p) throws IOException	接收一个 UDP 数据包。一个 UDP 数据包就是一个 DatagramPacket 对象
void connect(InetAddress address,int port)	将该 UDPSocket 变成一个连接型的 UDPSocket
void disconnect()	将该 UDPSocket 变成一个非连接型的 UDPSocket
void close()	关闭 UDPSocket

说明：UDPSocket 分为“连接型”与“非连接型”两种。默认情况下，UDPSocket 是“非连接型”的，这个“连接”不是指像 TCP 那样进行三步握手，而只是将对方（通常是 UDP 的 IP 地址与端口号）信息与自己的 UDPSocket 对象关联在一起。这样发送数据包时，内核要自动加上对方的 IP 地址与端口号。接收数据包时，内核可自动进行数据包过滤操作，不是来自于指定 IP 地址的数据包将被过滤掉。

DatagramPacket 类的对象代表了一个 UDP 数据报包。通过 UDP 发送数据时，先要根据发送的数据生成一个 DatagramPacket 对象，然后通过 DatagramSocket 对象的 send()方法发送这个对象。接收时，先要根据要接收数据的缓冲区生成一个 DatagramPacket 对象，然后通过 DatagramSocket 对象的 receive()方法接收这个对象的数据内容。

DatagramPacket 类的构造方法分为两类：一类是创建 DatagramPacket 对象用来接收数据报包；另一类是创建 DatagramPacket 对象用来发送数据报包。它们的区别是，用于发送数据报包的构造方法需要设定数据报到达的目的地址，若是“连接型”UDP，则不需要设定目的地址，而用于接收数据报包的构造方法无须设定地址。

- 用于接收数据报包的构造方法如下。

DatagramPacket(byte[] buf,int length)

功能：由接收缓冲区(byte[])字节数组与它的长度 length)生成一个 DatagramPacket 对象。buf 表示保存传入数据报的缓冲区，length 表示要读取的字节数。

DatagramPacket(byte[] buf,int offset,int length)

功能：构造 DatagramPacket，用来接收长度为 length 的包，在缓冲区中指定了偏移量。

- 用于发送数据报包的构造方法如下。

DatagramPacket(byte[] buf,int length, InetAddress address,int port)

功能：构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号。length 参数必须小于等于 buf.length。因为默认时 UDPSocket 是非连接型，故要在每一个发送用的 UDP 数据包中，除了含有发送的数据外，还要有接收方的 IP 地址和端口号。

DatagramPacket(byte[] buf,int offset,int length,InetAddress address,int port)

功能：构造数据报包，用来将长度为 length，偏移量为 offset 的包发送到指定主机上的指定端口号，length 参数必须小于等于 buf.length。

DatagramPacket 类的常用方法如表 13.5 所示。

表 13.5 DatagramPacket 类的常用方法

方 法	功 能
byte[] getData()	返回 DatagramPacket 对象中包含的数据
int getLength()	返回发送/接收数据的长度
int getOffset()	返回发送/接收数据在 byte[] 中的偏移
InetAddress getAddress()	返回对方的 IP 地址。用 InetAddress 对象表示
int getPort()	返回对方的端口号
void setData(byte[] buf,int offset,int length)	设置该 DatagramPacket 对象中包含的数据
void setAddress(InetAddress iaddr)	设置该 DatagramPacket 对象中包含的 IP 地址
void setPort(int iport)	设置该 DatagramPacket 对象中包含的端口号

13.3.1 UDP通信程序

【例 13.3】 一个简单 UDP 通信程序，客户端程序向服务器端程序发送任意的字符串，服务器端程序收到后，计算字符串的长度并向客户端程序回送相同的字符串。

首先建立一个 UDP 服务器端程序。在这个程序中，创建了一个在 9777 端口上等待接收数据报的 DatagramSocket 对象。当有数据报要接收时，创建一个 DatagramPacket 对象，接收此数据报。收到后分析数据报，并再次创建一个 DatagramPacket 对象，回送此数据报。

UDP 服务器端 UDPServer.java

```
package org;
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class UDPServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket(9777);           // 在 9777 端口上创建 UDPSocket
        Frame f = new Frame("UDPServer");
        f.setLocation(400, 300);
        f.setSize(300, 300);
        TextField tf = new TextField(40);
        TextArea list = new TextArea();
        f.add(list, BorderLayout.NORTH);
        f.pack();
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter() {               // 关闭窗口
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        byte[] buf = new byte[1024];                             // 接收缓冲区
        DatagramPacket dp = null;                                // 接收 UDP 数据包
        DatagramPacket sdp = null;                               // 发送 UDP 数据包
```

```

boolean more = true;                                // 控制 UDP 服务器
while (more) {
    dp = new DatagramPacket(buf, 1024);              // 创建一个用于接收的 UDP 数据包
    ds.receive(dp);                                  // 等待任一个客户机发送数据包
    InetAddress caddr = dp.getAddress();             // 获取客户机的 IP 地址
    int cport = dp.getPort();                         // 获取客户机的端口号
    // 获取客户机发送的文本内容
    String s = new String(dp.getData(), dp.getOffset(), dp.getLength());
    String str = "客户机 IP: " + caddr + " 客户机端口号: " + cport + "\n"
        + " 客户机发送的数据是: " + s + "\n";
    list.append(str);
    String rs = new String("字符串: " + s + " 的长度是: " + s.length());
    byte[] sbuf = rs.getBytes();                     // 将串转换成字节数组
    // 生成一个发送回特定客户机的 UDP 数据包的 DatagramPacket 对象
    sdp = new DatagramPacket(sbuf, sbuf.length, caddr, cport);
    ds.send(sdp);                                    // 向客户机发回响应信息
}
}
}

```

客户端程序创建一个 `DatagramSocket` 对象，准备发送数据。创建一个 `DatagramPacket` 对象把要发送的数据打包成一个数据包，并在数据包中写明要发送给远程服务器的 IP 地址与端口号。

UDP 客户端 UDPClient.java

```

package org;
import java.io.*;
import java.net.*;
import java.util.Date;
import java.awt.*;
import java.awt.event.*;
public class UDPClient {
    static List list = new List(6);
    static TextField tf = new TextField(40);
    static DatagramPacket sdp = null;
    static DatagramPacket rdp = null;
    public static void main(String[] args) throws Exception {
        Frame f = new Frame("UDPClient");
        DatagramSocket ds = new DatagramSocket();      // 生成一个客户机用 UDPSocket
        f.setLocation(400, 300);
        f.setSize(300, 300);
        f.add(tf, BorderLayout.SOUTH);
        f.add(list, BorderLayout.NORTH);
        f.pack();
        f.addWindowListener(new WindowAdapter() {      // 关闭窗口
            public void windowClosing(WindowEvent arg0) {
                System.exit(0);
            }
        })
    }
}

```

```

});
tf.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        try {
            byte[] rbuf = new byte[1024],           // 接收缓冲区大小设置为 1024
            sbuf = null;
            String str =tf.getText();               // 获取文本框的内容
            list.add(str);                          // 将数据添加到列表框中
            tf.setText(null);
            sbuf = str.getBytes();                  // 转换字节数组
            // 生成一个发送给 UDP 服务器的 UDP 数据包
            sdp = new DatagramPacket(sbuf,
                sbuf.length, InetAddress.getByName("127.0.0.1"), 9777);
            // 生成一个发送给 UDP 服务器的 UDP 数据包
            DatagramSocket ds = new DatagramSocket();
            ds.send(sdp);                          // 发送出去
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
});
f.setVisible(true);
ds.close();                                     // 关闭 Socket
}
}

```

运行上面的 UDP 客户端、服务器端程序。客户端程序向服务器端程序发送数据包，服务器端接收此数据包，显示收到的字符串内容，以及客户机的 IP 地址和端口号，如图 13.5、图 13.6 所示。

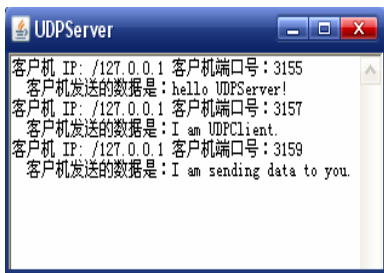


图 13.5 服务器端

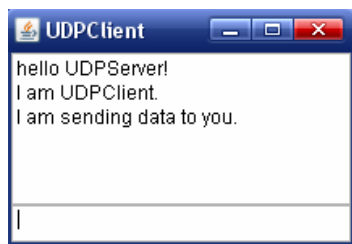


图 13.6 客户端

13.3.2 组播

IP 地址分 A、B、C、D、E 五类，D 类地址就是组播地址，从 224.0.0.0 到 239.255.255.255，其中 224.0.0.0 保留未用，组播通信是一对多的。因特网上的组播可以是本地级的，也可以是全局级的。在本地级，一个局域网上的一些主机可构成一个组，并指派一个组播地址。在全局级，不同网络上的主机可构成一个组，并指派一个组播地址。当然，全局级的组播需要路由器的支持。因特网中已经指派的一些组播地址有：224.0.0.1(这个子网的所有系统)、224.0.0.2

(这个子网上的所有路由器)、224.0.1.7 (AUDIONEWS 组)。

组播地址只能用做目的地址。若客户端对某一个组感兴趣，则可加入该组，成为该组中的一员。以后发送到该组的 UDP 数据报包，都自动传送到该组中的所有成员。客户端可以随时离开所加入的组，从而不再成为其中的一员，当然也就收不到发到该组的 UDP 数据报包了。Java 对组播有良好的支持，相关的类是 `java.net.MulticastSocket` 类。

`java.net.MulticastSocket` 类的对象，代表一个组播 `Socket`，可以发送和接收组播包。其实 `java.net.MulticastSocket` 类是 `java.net.DatagramSocket` 类的子类，只是增加了支持组播功能，因此 `DatagramSocket` 类中方法都可以使用。组播包就是一个 UDP 数据报包。

`MulticastSocket` 的构造方法如下。

● `MulticastSocket()` throws `IOException`

功能：创建一个组播 `Socket`。

● `MulticastSocket(int port)` throws `IOException`

功能：在指定端口上创建一个组播 `Socket`。

表 13.6 列出了 `MulticastSocket` 类的常用方法。

表 13.6 `MulticastSocket` 类的常用方法

方 法	功 能
<code>void joinGroup(InetAddress mcastaddr)</code> throws <code>IOException</code>	加入一个组，组地址是 <code>mcastaddr</code>
<code>void leaveGroup(InetAddress mcastaddr)</code> throws <code>IOException</code>	离开一个组，组地址是 <code>mcastaddr</code>
<code>void send(DatagramPacket p,byte ttl)</code> throws <code>IOException</code>	发送数据报包 <code>p</code> ，其生存期为 <code>ttl</code> ，通常是数据报包经过的路由器个数
<code>void setTimeToLive(int ttl)</code> throws <code>IOException</code>	设置组播数据包的生存期， $0 \leq ttl \leq 255$ 。若 <code>ttl</code> 为 2，则表示最多经过 2 个路由器，否则数据报包被丢弃

向一个组播组发送 UDP 数据报包，可以通过 `DatagramSocket`，像通常发送 UDP 数据报包一样进行发送，唯一区别是目的地址是一个组播地址。也可以通过 `MulticastSocket` 调用 `send()` 方法进行发送，好处是可以控制 UDP 数据报包的 `ttl`。调用 `joinGroup()` 加入一个组后，像通常接收 UDP 数据报包一样接收组播包。

说明：由于 UDP 是面向无连接的，故一个 UDP 服务器可同时接收所有客户端的数据报包并进行处理。因而 UDP 服务器通常可以不用像 TCP 服务器那样，采用多线程技术。同样，客户端收到的 UDP 数据报包，不能肯定一定是来自 UDP 服务器。若客户端不想对 UDP 数据报包的 IP 地址进行检验，则可采用“连接型” `UDPSocket`。

一个 UDP 数据报包理论上最大不能超过 64KB。在实际应用中，通常要小得多，比如 296KB 等。大的数据报包数据吞吐量大，但经过网络传输时，下层可能要拆分成许多小的部分，从而只要有一个小的部分丢失，则整个 UDP 数据报包最终被丢弃。但太小的 UDP 数据报包又使得传输效率差（每一个 UDP 数据报包中，UDP 报头要占用一些字节），故选择大小适宜的数据报包是有意义的。UDP 通信模型中，服务器与客户端可以是对等的。如通过设置 `UDPSocket`

超时选项，则双方运行的程序可以是同一个。这在 TCP 通信中几乎是不可能发生的。

13.4 URL

前面讨论的是基于 Socket 的 Java 网络编程，现在讨论建立在传输层 TCP 或 UDP 之上的应用层协议的 Java 网络编程。应用层协议很多，如 HTTP 协议、SMTP 协议、FTP 协议、POP3 协议等。本节讨论的是基于 HTTP 协议的 Java 网络编程。

URL（Uniform Resource Locator，统一资源定位器），用来表示从因特网上得到的资源位置和访问这些资源的方法。其格式是：协议名://资源名。其中协议名指明获取该资源所应使用的传输协议，如 http、ftp 等。资源名则是资源的完整位置，包括主机名、端口号、文件名或文件内部的一个引用，以及可能要传递的一些参数。并非所有的 URL 都包含这些内容，对于多数的协议，主机名和文件名是必需的，但端口号和文件内部的引用以及参数则是任选的。例如：http://www.njnu.edu.cn/java/index.htm，其中：协议名是 http 协议，主机名是 www.njnu.edu.cn，端口号是默认的 80，文件路径名是/java/index.htm。

浏览器与 Web 服务器是典型的基于 http 协议进行工作的 C/S 方式下的两个应用程序。浏览器向 Web 服务器发出 http 请求，Web 服务器处理该请求，并做出 http 应答。以 html 或 xml 等描述的网页内容，就包含在 http 应答中，并从 Web 服务器发送到浏览器，浏览器对 html 或 xml 进行解释，从而显示出网页。如果要在 Java 程序中读取 Web 页的内容，需要使用 java.net 包中的相关类。

类 java.net.URL 代表一个统一资源定位符，它是指向因特网资源的指针。资源可以是简单的文件或目录，也可以是对更为复杂的对象的引用，例如对数据库或搜索引擎的查询。

URL 类的构造方法如下。

● URL(String spec)throws MalformedURLException

功能：由一个表示 URL 地址的字符串构造一个 URL 对象。

● URL(String protocol, String host,int port, String file)throws MalformedURLException

功能：根据指定 protocol、host、port 号和 file 创建 URL 对象。

URL 类的常用方法如表 13.7 所示。

表 13.7 URL 类的常用方法

方 法	功 能
URLConnection openConnection()throws IOException	创建并返回一个 URLConnection 对象，它表示到 URL 所引用的远程对象的连接
final InputStream openStream() throws IOException	打开一个连接到该 URL 的 InputStream 的对象,通过该对象,可从 URL 中读取 Web 页面内容
final Object getContent() throws IOException	获取此 URL 的内容

【例 13.4】 创建一个 URL 对象，通过 URL，读取 www.sina.com 网页内容。

Download.java

```
package com.net;
import java.net.*;
```

```

import javax.swing.*;
import java.awt.event.*;
import java.io.*;
public class Download {
    public static void main(String[] args) {
        JFrame jf = new JFrame("下载程序");
        jf.setSize(600, 400);
        jf.setLocation(100, 100);
        JPanel p = new JPanel();
        JLabel l = new JLabel("Please input URL:");
        final JTextField tf = new JTextField(30);
        p.add(l);
        p.add(tf);
        jf.getContentPane().add(p, "North");
        final JTextArea ta = new JTextArea();
        jf.getContentPane().add(ta, "Center");
        JButton btn = new JButton("Download");
        jf.getContentPane().add(btn, "South");
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String str = tf.getText();           // 获取文本框中输入的 URL 地址
                try {
                    URL url = new URL(str);
                    // 产生一个缓冲流
                    BufferedReader br = new BufferedReader(new
                        InputStreamReader(url.openStream()));
                    String s;
                    while ((s = br.readLine()) != null){
                        ta.append(s+"\n");           // 将读取的内容添加到文本区中
                    }
                    br.close();                     // 关闭缓冲流
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        });
        jf.addWindowListener(new WindowAdapter() { // 关闭窗体
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        jf.setVisible(true);                       // 设置窗体可见
    }
}

```

运行程序，在地址栏中输入要访问的网址，例如谷歌的首页地址“<http://www.google.cn/>”，单击“Download”按钮，则谷歌的首页地址的源代码就下载到文本区里了，如图 13.7 所示。



图 13.7 使用 URL

13.5 综合实例

在这个综合实例中，将使用多线程的方式实现在线聊天室。程序可实现多人同时聊天。多个人的聊天记录都显示在每个人的窗体上。

思路分析：要实现多人同时聊天，首先服务器端要使用多线程的方式，服务器端的主进程不断监听客户端发起的连接。如果监听到客户端发起的连接请求，服务器端接收此连接请求，并创建一个从属进程接收客户端发送过来的数据。接收到数据之后，再转发到所有的客户端。

服务器端程序 ChatServer.java

```
package com.net;
import java.io.*;
import java.net.*;
import java.util.*;
public class ChatServer {
    boolean started = false;
    ServerSocket ss = null;
    // clients 是共享变量，通过 Collections.synchronized(...)做同步化处理
    List<Client> clients =Collections.synchronized( new ArrayList<Client>());
    /*synchronized 主要用于标识临界区，使得这些临界区成为一种逻辑上的原子操作，从而达到线程间的同步。*/
    public static void main(String[] args) {
        new ChatServer().start();
    }
    public void start() {
        try {
            ss = new ServerSocket(8888); // 创建一个监听 Socket 对象
            started = true;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    try {
        while (started) {
            Socket s = ss.accept();           // 等待客户端发起连接
            Client c = new Client(s);
            System.out.println("a client connected!");
            new Thread(c).start();           // 启动线程
            clients.add(c);                   // 向共享变量中添加
        }
        ss.close();                           // 关闭 Socket
    } catch (IOException e) {
        e.printStackTrace();
    }
}

class Client implements Runnable {           // 实现 Runnable 接口
    private Socket s;
    private DataInputStream dis = null;
    private DataOutputStream dos = null;
    private boolean Connected = false;
    public Client(Socket s) {
        this.s = s;
        try {
            dis = new DataInputStream(s.getInputStream()); // 创建输入流
            dos = new DataOutputStream(s.getOutputStream()); // 创建输出流
            Connected = true;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void send(String str) {
        try {
            dos.writeUTF(str);                // 向输入流中写入数据
        } catch (IOException e) {
            clients.remove(this);              // 出错时（客户可能已断线），
                                                // 移除一个客户端
        }
    }

    public void run() {
        try {
            while (Connected) {
                String str = dis.readUTF();    // 从输出流中读取数据
                /* 对共享的列表进行遍历时必须要同步化，对临界区使用对象锁来互
                   斥访问临界区资源 */
                synchronized(clients){
                    Iterator<Client> it = clients.iterator(); // 返回一个迭代器
                    while(it.hasNext()) {
                        Client c = it.next();

```

```

        c.send(str);                // 将数据发送出去
    } //while
} //synchronized
} //while(Connected)
dis.close();                      // 关闭输入流
dos.close();                      // 关闭输出流
s.close();                        // 关闭 Socket
} catch (Exception e) {
    System.out.println("Client closed!");
}
finally{
    clients.remove(this);
    // 确保线程结束时从共享变量中删除自己(比如从客户机读数据时出错, 客户机可能已掉线,
    // 线程会结束)
} //try
} //run
}
}

```

客户端程序 ChatClient.java

```

package com.net;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;

public class ChatClient extends Frame {
    static Icon[] icons = new Icon[] {
        new ImageIcon("E:/workbench/MyProject_13/src/com/net/image0.jpg"),
        new ImageIcon("E:/workbench/MyProject_13/src/com/net/image1.jpg"), };
    Socket s = null;
    DataOutputStream dos = null; DataInputStream dis = null;
    private boolean Connected = false;
    TextField tf = new TextField();
    TextArea ta1 = new TextArea();
    TextArea ta2 = new TextArea();
    Button bt1 = new Button("发送");
    Thread thread = new Thread(new ClientThread()); // 创建线程
    public static void main(String[] args) {
        new ChatClient().call();
    }
    public void call() {
        bt1.setBackground(Color.cyan);
        JButton jt1 = new JButton(icons[0]);
        JButton jt2 = new JButton(icons[1]);
    }
}

```

```

        jt1.setBounds(265, 40, 80, 80);
        jt2.setBounds(265, 140, 80, 80);
        setLocation(400, 300);
        setSize(400, 300);
        setLayout(null); // 取消布局管理器
        setBackground(Color.cyan);
        tf.setBounds(250, 40, 70, 25);
        ta1.setBounds(30, 40, 200, 80);
        ta2.setBounds(30, 140, 200, 80);
        bt1.setBounds(265, 250, 70, 30);
        tf.setBounds(30, 240, 200, 35);
        tf.addActionListener(new MyListener()); // 注册事件监听器
        add(tf);add(jt1);add(jt2);add(bt1);
        add(ta1);add(ta2);add(tf);
        this.addWindowListener(new WindowAdapter() { // 关闭窗口
            public void windowClosing(WindowEvent e) {
                disconnect();
                System.exit(0);
            }
        });
        bt1.addActionListener(new MyListener()); // 注册事件监听器
        setVisible(true);
        connect();
        thread.start(); // 启动线程
    }

    public void connect() {
        try {
            s = new Socket("127.0.0.1", 8888);
            dos = new DataOutputStream(s.getOutputStream()); // 返回一个输出流
            dis = new DataInputStream(s.getInputStream()); // 返回一个输入流
            System.out.println("connected!");
            Connected = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void disconnect() {
        try {
            dos.close(); // 关闭输出流
            dis.close(); // 关闭输入流
            s.close(); // 关闭 Socket
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private class MyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {

```

```

String str = tf.getText().trim();           // 获取文本框中的数据
tf.setText("");
ta2.append(str+"\n");                       // 将文本框中的数据添加到文本区中
try {
    dos.writeUTF(str);                     // 向输出流中写入数据
    dos.flush();                          // 刷新输出流
} catch (IOException e1) {
    e1.printStackTrace();
}
}
}

private class ClientThread implements Runnable {
    public void run() {
        try {
            while (Connected) {
                String str = dis.readUTF(); // 从输出流中读取数据
                ta1.append(str+"\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}

```

首先运行服务器程序 ChatServer.java，接着启动两个客户端程序，出现两个客户端窗口。两个客户端之间就可以互相发送数据，在各自的窗口上显示发送的数据和接收的数据，如图 13.8、图 13.9 所示。

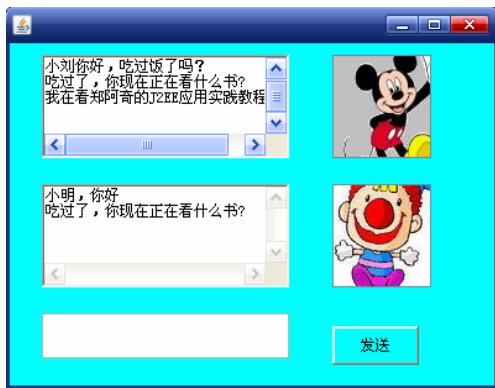


图 13.8 一个客户端 (1)

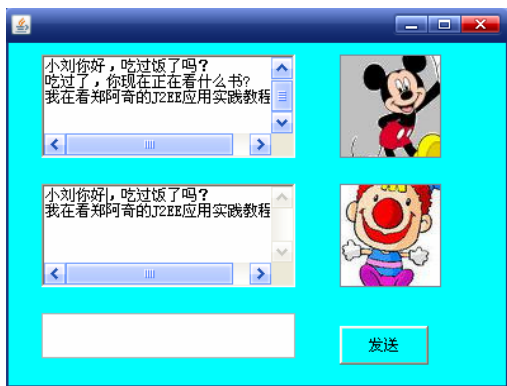


图 13.9 一个客户端 (2)

第 14 章 JDBC编程

在实际的项目开发中，应用程序通常需要同后台的数据库连接进行查询、更新等操作。不同平台有不同的连接数据库的应用编程接口。在 Java 平台中，连接数据库的应用编程接口就是 JDBC（Java DataBase Connectivity）。Java 应用程序可以通过 JDBC 来连接数据库。

JDBC 是一种可用于执行 SQL 语句的 Java API，主要提供了跨平台的数据库访问方法，为数据库应用开发人员提供了一种标准的应用程序编程接口，类似于 Microsoft 的 ODBC。但 ODBC 只针对 Windows 平台，而且 ODBC 要求在客户机上安装和注册，因而维护成本比较高。相对而言，JDBC 的维护和部署要简单得多，因为它是真正的跨平台的 API，屏蔽了具体数据库的差异性，如图 14.1 所示。当 Java 程序访问数据库时，由 JDBC API 接口调用相应数据库的 JDBC API 实现来访问数据库，从而无须改变 Java 程序就能访问不同的数据库。JDBC 有不同的版本，本书介绍最新的版本 JDBC 4.0。

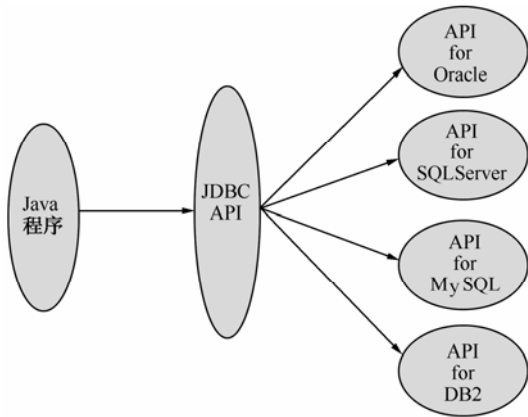


图 14.1 Java 程序访问数据库

14.1 SQL语言

SQL（Structured Query Language）是一个标准的结构化的数据库查询语言，最初是由 IBM 公司提出的，目的是为了在不同的数据库之间构建一个统一的操作平台。由于 SQL 具有结构化、简单易学且功能全面的特点，1987 年被 ANSI 制定为标准。此后，几乎所有的关系数据库都支持 SQL。因此，学习、使用 JDBC 编程 Java 程序，必须了解常用的 SQL 语句。SQL 语言有近百条语句，下面介绍几条最常用、最基本的 SQL 语句。

1. 查询语句

格式：

```
select col1, col2, ..., coln from table_name [where condition_expression]
```

描述：其中参数 col1、col2 等均为列名，table_name 为表名，condition_expression 为条件表达式。select、from、where 都是保留字。

功能：从数据库表中检索满足条件的记录。例如下面的语句：

```
select * from student;
```

表示将表 student 中的全部数据检索出来。这里 “*” 代表所有列。

```
select id,name,math,english from student where math+english>80 and math+english<100;
```

表示将表 student 中满足 math 与 english 之和在 80~100 之间的数据检索出来。其中 and 是保留字，表示逻辑操作 “与”。类似的还有 not、or。

```
select * from student where name like '王%';
```

表示将表 student 中所有姓王的同学的数据检索出来。其中 like 是保留字，表示字符串比较，“%” 代表任意的字符串。

2. 插入语句

格式：

```
insert into table_name [(col1,col2,...,coln)] values(v1,v2,...,vn)
```

功能：在表 table_name 中插入一条记录，各列的值依次分别为 v1,v2,...,vn 等，若某列的列名未给，则值为 NULL。其中 insert、into、values 都是保留字。

注意：若所有的列名都未给，则在 values 中必须依次给出所有列的值。给出的值的类型必须与对应的列的类型相一致。例如下面的语句：

```
insert into student values('001108','刘中华','男',95,94)
```

表示在表 student 中插入一条记录，各列的值依次为 “001108”、“刘中华”、“男”、95、94。该表只有五列。

```
insert into student(id,math,english)values('001109',96,91)
```

表示在表 student 中插入一条记录，其 id 列的值为 ‘001109’，math 列的值为 96，english 列的值为 91，其余列的值均为 null。

3. 更新语句

格式：

```
update table_name set col1=v1 [,col2=v2,...,coln=vn][where condition_expression]
```

功能：更新表 table_name 中满足条件的记录，使列 col1 的值为 v1，列 col2 的值为 v2……列 coln 的值为 vn 等。其中 update、set、where 都是保留字。

注意：如不给出条件，则更新表中所有记录。例如下面的语句：

```
update student set math=0,english=0
```

表示将表 student 中所有记录的 math 列、english 列的值变为 0。

```
update student set math=0 where sex='男'
```

表示将表 student 中满足 sex=‘男’的记录的 math 值置为 0。

```
update student set english=0 where name like '刘%'
```

表示将表 student 中所有姓刘的记录的 english 值置为 0。

4. 删除语句

格式:

```
delete from table_name [where condition_expression]
```

功能: 删除表 `table_name` 中满足条件的记录。其中 `delete`、`from`、`where` 都是保留字。

特别注意: 如果不给出条件, 则删除表中所有的记录。例如下面的语句:

```
delete from student where math+english<30;
```

表示删除表 `student` 中所有满足 “`math+english<30`” 的记录。

5. 建表语句

格式:

```
create table table_name (col1 type [not] null,...)
```

功能: 在当前数据库中创建一张名为 `table_name` 的表格, 其中 `create`、`table`、`not`、`null` 等都是保留字。例如下面的语句:

```
create table stuinfo(id char(10) not null,name char(10) null,height integer null)
```

表示在当前数据库中创建一个名为 `stuinfo` 的表, 有三列: 第一列列名是 `id`, 类型是 `char`, 宽度为 10, 非空; 第二列列名是 `name`, 类型是 `char`, 宽度为 10, 可以为空; 第三列列名是 `height`, 类型是 `integer`, 可以为空。

6. 删除表

格式:

```
drop table_name
```

功能: 在当前数据库中删除名为 `table_name` 的表, 其中 `drop` 是保留字。例如下面的语句:

```
drop stuinfo;
```

表示从当前数据库中删除表 `stuinfo`。

14.2 JDBC

JDBC API 为开发者使用数据库提供了统一的接口, 使得 Java 程序无须关心具体数据库的差异性, 这是通过 JDBC API 实现获得的。不同的数据库提供不同的 JDBC API 实现, 如图 14.2 所示。JDBC 的实现包括以下三部分。

- JDBC 驱动管理器: `java.sql.DriverManager` 类, 负责注册特定 JDBC 驱动器, 以及根据特定驱动器建立与数据库的连接。

- JDBC 驱动器 API: 其中最主要的接口是 `java.sql.Driver` 接口。

- JDBC 驱动器: 由数据库供应商或其他第三方工具提供商创建, 也称为 JDBC 驱动程序。JDBC 驱动器实现了 JDBC 驱动器 API, 负责与特定的数据库连接。JDBC 驱动器可以注册到 JDBC 驱动管理器中。不同的数据库提供不同的 JDBC 驱动器。

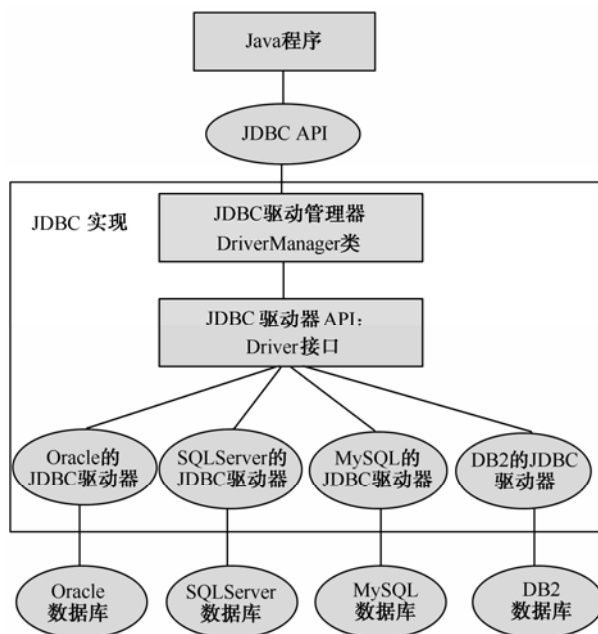


图 14.2 JDBC 实现

JDBC 驱动器有以下四种类型。

(1) JDBC-ODBC 驱动程序 (称为: Type1 型驱动程序)

该驱动程序首先将对 JDBC 的调用转化为 ODBC 的调用, 然后再利用 ODBC 与数据库进行连接, 如图 14.3 所示。这要求本地必须安装 ODBC 驱动程序, 然后注册一个 ODBC 数据源, 显然性能不是很高, 而且还限制了可移植性, 但在 Windows 环境中比较常用, 因为 Windows 本身自带 ODBC, 而且几乎所有的数据库都支持 ODBC。SUN 公司提供的 JDBC-ODBC 桥驱动程序就属于 Type1 型。

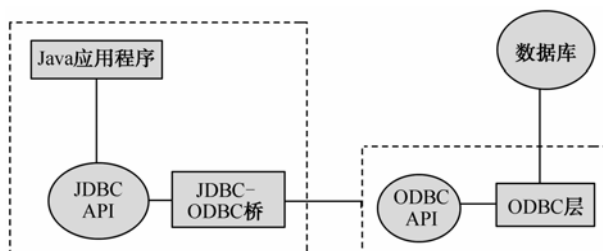


图 14.3 通过 JDBC-ODBC 访问数据库

(2) 本地代码和 Java 驱动程序 (称为: Type2 型驱动程序)

在程序中利用 JDBC API 访问数据库时, JDBC 驱动程序将调用请求转换为厂商提供的本地 API 调用, 如图 14.4 所示。这要求本地必须安装好特定的驱动程序, 显然限制了应用程序对其他数据库的使用, 这种方式很少使用。

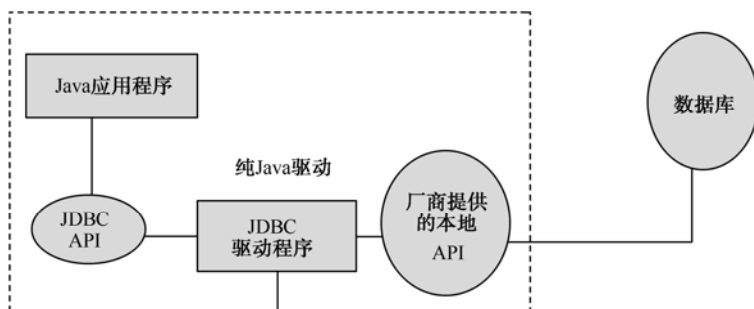


图 14.4 利用厂商提供的本地 API 访问数据库

(3) JDBC 网络纯 Java 驱动程序（称为：Type3 型驱动程序）

它是独立于数据库服务器的，它和一个中间件服务器通信，由中间件负责与数据库通信。在本地机不需要安装任何软件，但在服务器端必须安装中间件软件，如图 14.5 所示。中间件提供了灵活性，可以用相同的代码访问不同的数据库，比较适合异构数据库的应用，但也容易降低性能。Java 应用程序通过 JDBC 驱动程序将调用发送给应用服务器，应用服务器使用本地驱动程序访问数据库，完成应用请求。

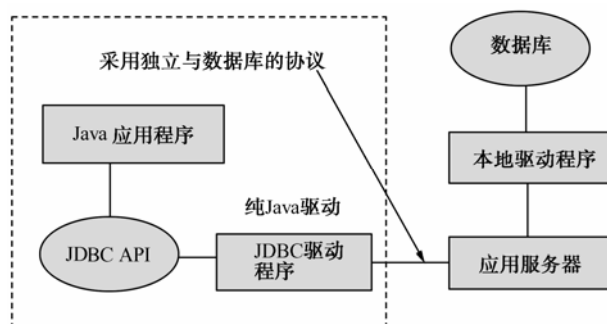


图 14.5 利用中间件的服务器访问数据库

(4) 本地协议的纯 Java 驱动程序（称为：Type4 型驱动程序）

使用该类型的应用程序无须安装附加的软件，所有对数据库的操作都直接由 JDBC 驱动程序完成，它将用户的请求直接转换为对数据库的协议请求，如图 14.6 所示。这种方式不会增加任何额外的负担，显然这种类型提供了最佳的性能。

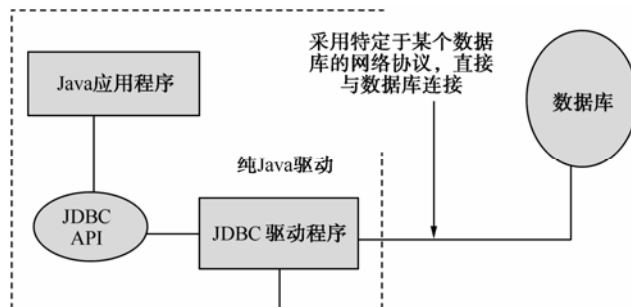


图 14.6 利用本地协议的纯 Java 驱动程序访问数据库

14.3 MySQL数据库

MySQL 是一个多用户、多线程的数据库，开发者为瑞典 MySQL AB 公司。目前 MySQL 被广泛地应用在 Internet 上的中小型网站中。由于其体积小，速度快，总体拥有成本低，尤其是开放源码这一特点，许多中小型网站为了降低网站总体拥有成本而选择了 MySQL 作为网站数据库。MySQL 的官方网站的网址是：www.mysql.com。为方便读者调试程序，本章中的实例采用 MySQL 数据库。MySQL 服务器安装包可以从 <http://dev.mysql.com/downloads/mysql/5.0.html> 上免费下载。目前最新的版本是 MySQL 6.0 测试版，本书安装的是 MySQL 5.0，安装环境是在 Windows 操作系统下。由于本章要用到 MySQL 的 JDBC 驱动程序（MySQL Connector/J），也一并下载下来。

14.3.1 MySQL服务器的安装

MySQL 的安装比较简单，步骤如下。

- （1）下载 Windows 版的 MySQL 5.0，解压后双击进入安装向导。选择 Typical，单击“Next”按钮进入下一步。
- （2）进入如图 14.7 所示界面，在 MySQL 5.0 中，默认目录为 C:\Program Files\MySQL\MySQL Server 5.0。确认后单击“Install”按钮开始安装。安装完成后出现创建一个 MySQL.com 账号的画面。选择“Skip Sign-Up”，单击“Next”按钮，跳过这一步，最后单击“Finish”按钮完成安装。



图 14.7 MySQL 安装

14.3.2 MySQL服务器的配置

MySQL 安装完成后，要对 MySQL 服务器进行配置，步骤如下。

- （1）安装完毕后选择“Config the MySQL server now”复选框进入配置向导。选择“Detailed Configuration”的配置类型。
- （2）单击“Next”按钮进行服务器类型选择，这里选择“Developer Machine”。
- （3）单击“Next”按钮进入数据库使用情况对话框，这里选择“Multifunctional Database”。
- （4）进入 InnoDB 表空间对话框，这里可以修改 InnoDB 表空间文件的位置，如图 14.8 所示。默认位置是 MySQL 服务器数据目录，这里不做修改，直接进入下一步。

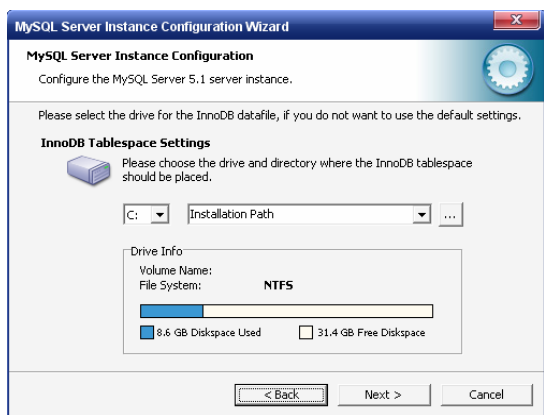


图 14.8 表空间对话框

(5) 接下来进入并发连接选择对话框，这里选择“Decision Support (DSS)/OLAP”。

(6) 进入联网选项对话框，如图 14.9 所示。默认情况是启用 TCP/IP 网络，默认端口为 3306，这里不做修改，直接进入下一步。

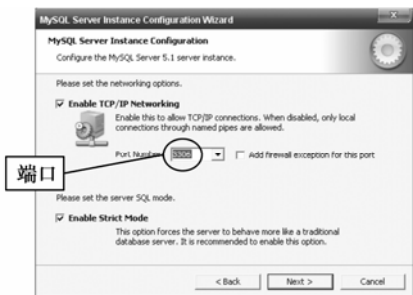


图 14.9 联网选项对话框

(7) 进入字符集选择对话框，前面的选项一直是按默认设置进行的，这里要做一些修改。选中“Manual Selected Default Character Set/Collation”选项，在“Character Set”文本框中将 latin1 修改为 gbk，如图 14.10 所示。

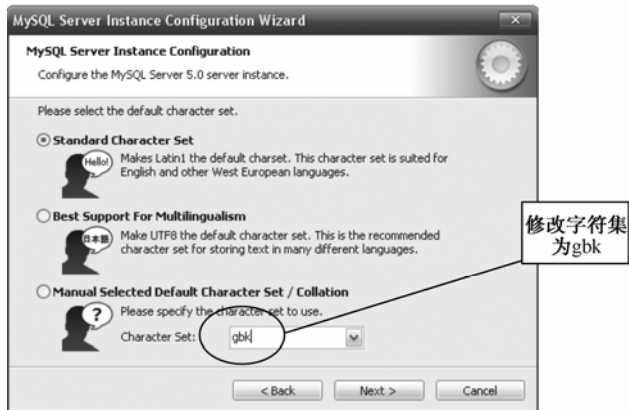


图 14.10 设置字符集

(8) 进入服务选项对话框，服务名为 MySQL，这里不做修改。

(9) 进入安全选项对话框，如图 14.11 所示，在密码输入框中输入 root 用户密码。



图 14.11 安全选项对话框

(10) 设置完毕后，进入提交配置对话框，单击“Execute”按钮即可完成。
以上步骤完成后，MySQL 服务器就可以使用了。

14.3.3 MySQL的环境

MySQL 安装和配置完成后，打开“开始”→“程序”→“MySQL”→“MySQL Server 5.0”→“MySQL Command Line Client”，就进入到 MySQL 客户端，在客户端窗口输入密码，就以 root 用户身份登录到 MySQL 服务器，在窗口中出现命令行，如图 14.12 所示，在命令行中输入 SQL 语句就可以操作 MySQL 数据库。以 root 用户身份登录可以对数据库进行所有的操作。

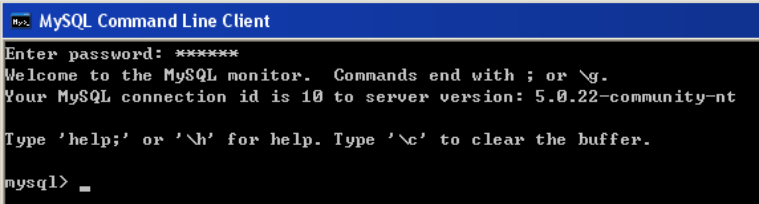


图 14.12 MySQL 命令行

MySQL 有一个 data 目录，用于存放数据库文件，其默认的路径为：C:\Program Files\MySQL\MySQL Server 5.0\data。在 data 目录中 MySQL 为每一个数据库建立一个文件夹，所有的表文件存放在相应的数据库文件夹中。

使用 MySQL 非常简单，这里介绍 MySQL 的常用命令，如表 14.1 所示。

表 14.1 MySQL 的常用命令

命 令	格 式	示 例
连接到 MySQL	MySQL -h 主机地址 -u 用户名 -p 密码	MySQL -h127.0.0.1 -uroot -p123456
退出 MySQL	exit	exit
创建数据库	create database 数据库名	create database xscj
创建表	use 数据库名 create table 表名 (字段列表)	create table xs(id int(6) not null primary key,name char(10)) not null,profession char(20) not null)
删除数据库	drop database 数据库名	drop database xscj

命 令	格 式	示 例
删除表	drop table 表名	drop table xs
插入记录	insert into 表名 values (字段值列表)	insert into xs(100001,'王军','计算机')
查询记录	select 字段列表 from 表名 where 约束条件	select name from xs where id = 100001
删除记录	delete from 表名 where 约束条件	delete from xs where id = 100002
修改记录	update 表名 set 列名 = 值 where 约束条件	update xs set name = '王涛' where id = 100002

现在在 MySQL 命令行窗口上使用这些命令。打开“开始”→“程序”→“MySQL”→“MySQL Server 5.0”→“MySQL Command Line Client”，就进入到 MySQL 客户端，在客户端窗口输入密码，就以 root 用户身份登录到 MySQL 服务器，输入如图 14.13 所示的命令，即可完成数据库的创建，表的创建，插入记录和查询记录。

```
mysql> create database xscj;
Query OK, 1 row affected (0.02 sec)

mysql> use xscj
Database changed
mysql> create table xs
-> (
-> id int(6) not null primary key,
-> name char(10) not null,
-> profession char(20) not null
-> );
Query OK, 0 rows affected (0.03 sec)

mysql> insert into xs values (100001, '王军','计算机');
Query OK, 1 row affected (0.05 sec)

mysql> select name from xs where id = 100001;
+-----+
| name |
+-----+
| 王军 |
+-----+
```

图 14.13 使用 MySQL 命令

14.4 访问数据库

在 Java 中，访问数据库的基本步骤如下。

- (1) 加载驱动程序。DriverManager 类是驱动程序管理器类，负责管理驱动程序。DriverManager 类的 registerDriver()方法用来注册驱动程序类的实例。
- (2) 建立连接。加载驱动程序后，调用 DriverManager 类的 getConnection()方法得到一个与数据库的连接，返回一个 Connection 对象。
- (3) 访问数据库，即执行 SQL 语句。得到数据库的连接后，就可以访问数据库了。调用 Connection 对象的 createStatement()、prepareStatement()方法来执行 SQL 语句，返回结果集。
- (4) 处理结果集，最后关闭结果集，断开连接。

以上这些操作都是通过调用相应类的方法来实现的，JDBC API 由 java.sql 和 javax.sql 包组成。java.sql 包定义了访问数据库的接口和类，如图 14.14 所示。下面介绍一些最常用的类及其方法。

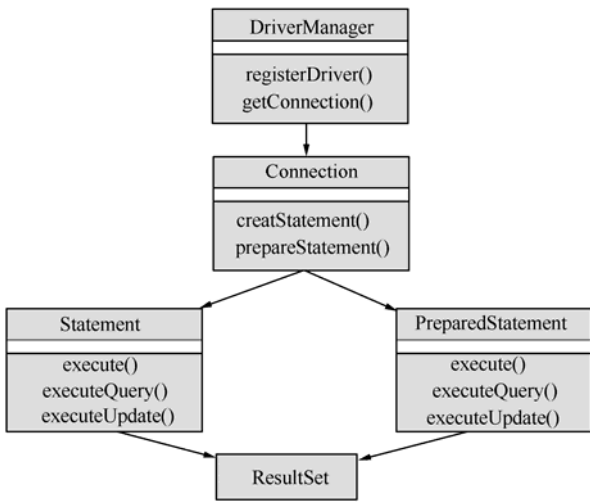


图 14.14 java.sql 包中主要的类和接口

14.4.1 加载并注册数据库驱动

1. Driver接口

java.sql.Driver 是所有 JDBC 驱动程序需要实现的接口，下面是不同数据库实现该接口的驱动程序类名。

- oracle.jdbc.driver.OracleDriver

这是 Oracle 数据库的 JDBC 驱动程序的类名，Oracle 的 JDBC 驱动不需要单独下载，它位于安装文件的 lib 目录下。

- com.microsoft.jdbc.sqlserver.SQLServerDriver

- com.microsoft.sqlserver.jdbc.SQLServerDriver

这两个都是 SQLServer 数据库的 JDBC 驱动类名。只不过上面的是 SQLServer 2000 的 JDBC 驱动类名，下面的是 SQLServer 2005 的 JDBC 驱动类名。

- com.mysql.jdbc.Driver

这是 MySQL 的 JDBC 驱动类名。

Driver 接口中提供了一个 Connect()方法，用来建立到数据库的连接。

Connection connect(String url,Properties info)throws SQLException

功能：试图创建一个到给定 URL 的数据库连接。

在程序中不需要直接调用访问这些实现了 Driver 接口的类，而是由驱动程序管理器去调用这些驱动。通过 JDBC 驱动程序管理器注册每个驱动程序，使用驱动程序管理器提供的方法来建立数据库连接，而驱动程序管理器类的连接方法则调用驱动程序类的 connect()方法建立数据库连接。

2. 加载与注册JDBC驱动器

加载 JDBC 驱动是调用 Class 类的静态方法 forName()，向其传递要加载的 JDBC 驱动类名。在运行时，类加载器从 CLASSPATH 环境变量中定位和加载 JDBC 驱动类。在加载驱

动程序类后，需要注册驱动程序类的一个实例。

`DriverManager` 类是驱动程序管理器类，负责管理驱动程序，这个类中的所有方法都是静态的。在 `DriverManager` 类中提供了 `registerDriver()` 方法来注册驱动程序类的实例。

```
static void registerDriver(Driver driver)throws SQLException
```

功能：向 `DriverManager` 注册给定驱动程序。`driver` 是将向 `DriverManager` 注册的新的 JDBC Driver。

```
static void setLoginTimeout(int seconds)
```

功能：设置驱动程序试图连接到某一数据库时将等待的最长时间，以秒为单位。

14.4.2 建立到数据库的连接

调用 `DriverManager` 类的 `getConnection()` 方法建立到数据库的连接，返回一个 `Connection` 对象。`Connection` 接口负责维护 Java 应用程序与数据库之间的连接。

```
public static Connection getConnection(String url, String user,String password)
throws SQLException
```

功能：试图建立到给定数据库 URL 的连接。`user` 是用户名，`password` 是用户的密码。其中 `url` 的形式为：

```
url = "jdbc:subprotocol:data source identifier"
```

`subprotocol` 表示与特定数据库系统相关的子协议，`data source identifier` 表示数据源信息。

对于 Oracle 数据库连接，其 `url` 的形式为：

```
url = "jdbc:oracle:thin:@localhost:1521:xscj"
```

对于 `SQLServer` 数据库连接，其 `url` 有以下两种形式为。

```
url = "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName = xscj"
```

```
url = "jdbc:sqlserver://localhost:1433;DatabaseName = xscj"
```

上面的 `url` 是对应于 `SQLServer 2000` 数据库的，下面的 `url` 是对应于 `SQLServer 2005` 数据库的。

对于 `MySQL` 数据库连接，其 `url` 的形式为：

```
url = "jdbc:mysql://localhost:3306/xscj"
```

`Connection` 接口的常用方法如下。

(1) `Statement createStatement()`throws `SQLException`

功能：创建一个 `Statement` 对象来将 SQL 语句发送到数据库。

(2) `PreparedStatement prepareStatement(String sql)` throws `SQLException`

功能：创建一个 `PreparedStatement` 对象来将参数化的 SQL 语句发送到数据库。不带参数的 SQL 语句通常使用 `Statement` 对象执行。如果多次执行相同的 SQL 语句，使用 `PreparedStatement` 对象可能更有效。

(3) `void commit()`throws `SQLException`

功能：使所有上一次提交/回滚后进行的更改成为持久更改，并释放此 `Connection` 对象当前持有的所有数据库锁。

(4) `void setAutoCommit(boolean autoCommit)`throws `SQLException`

功能：将此连接的自动提交模式设置为给定状态。如果连接处于自动提交模式下，则它的所有 SQL 语句将被执行并作为单个事务提交。否则，它的 SQL 语句将聚集到事务中，直到调用 commit()方法或 rollback()方法为止。默认情况下，新连接处于自动提交模式。

(5) void rollback()throws SQLException

功能：取消在当前事务中进行的所有更改，并释放此 Connection 对象当前持有的所有数据库锁。

(6) boolean isReadOnly()throws SQLException

功能：查询此 Connection 对象是否处于只读模式。

(7) boolean isClosed()throws SQLException

功能：查询此 Connection 对象是否已经被关闭。

(8) void close()throws SQLException

功能：立即释放此 Connection 对象的数据库和 JDBC 资源。

14.4.3 访问数据库

在与数据库建立连接后，需要对数据库进行访问，执行 SQL 语句。java.sql 包提供访问数据库的接口有：Statement、PreparedStatement、ResultSet。它们的作用是将 SQL 指令发送给数据库，并返回执行结果。

1. Statement接口

调用 Connection 对象的 createStatement()方法创建一个 Statement 对象。Statement 接口的常用方法如下。

(1) boolean execute(String sql)throws SQLException

功能：执行给定的 SQL 语句。

(2) int executeUpdate(String sql)throws SQLException

功能：执行给定 SQL 语句，该语句可能为 INSERT、UPDATE 或 DELETE 语句，或者不返回任何内容的 SQL 语句（如 SQL DDL 语句）。

(3) ResultSet executeQuery(String sql)throws SQLException

功能：执行给定的 SQL 语句，该语句返回单个 ResultSet 对象。

(4) void addBatch(String sql)throws SQLException

功能：将给定的 SQL 命令添加到此 Statement 对象的当前命令列表中。通过调用方法 executeBatch 可以批量执行此列表中的命令。

(5) int[] executeBatch()throws SQLException

功能：将一批命令提交给数据库来执行，如果全部命令执行成功，则返回更新计数组成的数组。

2. PreparedStatement接口

Java 提供了一个 Statement 接口的子接口 PreparedStatement，两者的功能相似，但当某 SQL 指令被执行多次时，PreparedStatement 的效率要比 Statement 高，而且 PreparedStatement 还可以给 SQL 指令传递参数。调用 Connection 对象的 prepareStatement()方法来得到

PreparedStatement 对象。PreparedStatement 对象所代表的 SQL 语句中的参数用问号 (?) 来表示。调用 PreparedStatement 对象的 setXXX()方法来设置这些参数。

PreparedStatement 接口的常用方法如下。

(1) void setBoolean(int parameterIndex,boolean x)throws SQLException

功能：将指定参数设置为给定 boolean 值。parameterIndex 的第一个参数是 1，第二个参数是 2……x 是参数值。

(2) void setInt(int parameterIndex,int x)throws SQLException

功能：将指定参数设置为给定 int 值。

(3) void setFloat(int parameterIndex,float x)throws SQLException

功能：将指定参数设置为给定 float 值。

(4) void setDouble(int parameterIndex,double x)throws SQLException

功能：将指定参数设置为给定 double 值。

(5) void setString(int parameterIndex,String x)throws SQLException

功能：将指定参数设置为给定 String 值。

(6) void setDate(int parameterIndex,Date x)throws SQLException

功能：使用运行应用程序的虚拟机的默认时区将指定参数设置为给定 java.sql.Date 值。

3. ResultSet接口

当使用 Statement 和 PreparedStatement 中的 executeQuery()方法来执行 select 查询指令时，查询的结果被放在结果集 ResultSet 中。

ResultSet 接口的常用方法如下。

(1) String getString(int columnIndex)throws SQLException

功能：获取此 ResultSet 对象的当前行中指定列的值，参数 columnIndex 代表字段的索引位置。

(2) String getString(String columnLabel)throws SQLException

功能：获取此 ResultSet 对象的当前行中指定列的值，参数 columnLabel 代表字段值。

(3) int getInt(int columnIndex)throws SQLException

功能：获取此 ResultSet 对象的当前行中指定列的值，参数 columnIndex 代表字段值。

(4) int getInt(String columnLabel)throws SQLException

功能：获取此 ResultSet 对象的当前行中指定列的值，参数 columnLabel 代表字段值。

(5) boolean absolute(int row)throws SQLException

功能：将光标移动到此 ResultSet 对象的给定行编号。

(6) boolean previous()throws SQLException

功能：将光标移动到此 ResultSet 对象的上一行。

(7) boolean first()throws SQLException

功能：将光标移动到此 ResultSet 对象的第一行。

(8) boolean last()throws SQLException

功能：将光标移动到此 ResultSet 对象的最后一行。

(9) boolean next()throws SQLException

功能：将游标移到下一行，ResultSet 游标最初位于第一行之前，第一次调用 next()方法使第一行成为当前行。

14.5 JDBC编程

现在就开始编写一个使用后台数据库为 MySQL 的 JDBC 程序。这需要在当前的工程中引入 MySQL 的 JDBC 驱动程序。方法是：右击工程“MyProject_14”，选择“Build Path”→“Add External Archives”，出现一个文件选择对话框，如图 14.15 所示，找到磁盘上的 MySQL 的 JDBC 驱动程序，选中它，单击“打开”按钮，MySQL 的 JDBC 驱动程序就被加入到当前的工程中。

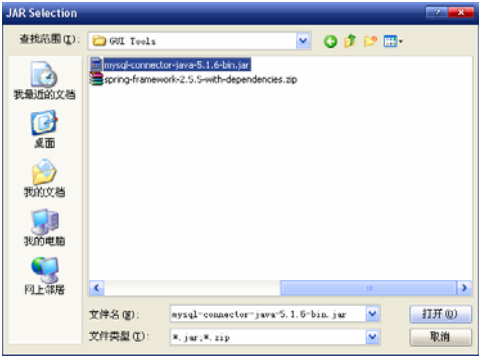


图 14.15 引入 MySQL 的 JDBC 程序

【例 14.1】 编写 JDBC 应用程序，使用已创建的 xscj 数据库和其中的 xs 表。

TestJDBC.java

```
package org.jdbc;
import java.sql.*;
public class TestJDBC {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        try {
            /* 加载并注册 MySQL 的 JDBC 驱动 */
            Class.forName("com.mysql.jdbc.Driver");
            /* 建立到 MySQL 数据库的连接*/
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            /* 访问数据库，执行 SQL 语句 */
            stmt = conn.createStatement();
            rs = stmt.executeQuery("select * from xs");
            while(rs.next()) {
                System.out.println(rs.getInt("id"));
                System.out.println(rs.getString("name"));
                System.out.println(rs.getString("profession"));
            }
        }
    }
}
```

```

    }
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if(rs != null) {
            rs.close();           // 关闭 ResultSet 对象
            rs = null;
        }
        if(stmt != null) {
            stmt.close();         // 关闭 Statement 对象
            stmt = null;
        }
        if(conn != null) {
            conn.close();         // 关闭 Connection 对象
            conn = null;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}
}

```

程序运行结果:

```

100001
王军
计算机

```

【例 14.2】 向数据库 xscj 中的 xs 表中添加记录、修改记录、删除记录。

TestDML.java

```

package org.jdbc;
import java.sql.*;

public class TestDML {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        try {
            /* 加载并注册 MySQL 的 JDBC 驱动 */
            Class.forName("com.mysql.jdbc.Driver");
            /* 建立到 MySQL 数据库的连接 */
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            /* 访问数据库, 执行 SQL 语句 */
            stmt = conn.createStatement();
            /* 添加记录 */

```

```

System.out.println("添加记录后:");
String sql1 = "insert into xs values (100002, '张艳', '电子信息工程');
stmt.executeUpdate(sql1);
rs = stmt.executeQuery("select * from xs");
while(rs.next()) {
    System.out.println(rs.getInt("id"));
    System.out.println(rs.getString("name"));
    System.out.println(rs.getString("profession"));
}
/* 修改记录 */
System.out.println("修改记录后:");
String sql2 = "update xs set name = '王涛' where id = 100002";
stmt.executeUpdate(sql2);
rs = stmt.executeQuery("select * from xs");
while(rs.next()) {
    System.out.println(rs.getInt("id"));
    System.out.println(rs.getString("name"));
    System.out.println(rs.getString("profession"));
}
System.out.println("删除记录后:");
/* 删除记录 */
String sql3 = "delete from xs where id = 100002";
stmt.executeUpdate(sql3);
rs = stmt.executeQuery("select * from xs");
while(rs.next()) {
    System.out.println(rs.getInt("id"));
    System.out.println(rs.getString("name"));
    System.out.println(rs.getString("profession"));
}
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if(rs != null) {
            rs.close();
            rs = null;
        }
        if(stmt != null) {
            stmt.close();
            stmt = null;
        }
        if(conn != null) {
            conn.close();
            conn = null;
        }
    }
}

```

// 关闭 ResultSet 对象

// 关闭 Statement 对象

// 关闭 Connection 对象

```

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}

```

程序运行结果：

```

添加记录后：
100001
王军
计算机
100002
张艳
电子信息工程
修改记录后：
100001
王军
计算机
100002
王涛
电子信息工程
删除记录后：
100001
王军
计算机

```

说明：参数不同的同一条 SQL 语句，可以使用比较高效的 `PreparedStatement` 对象。它表示一条预编译过的 SQL 语句。`PreparedStatement` 对象所代表的 SQL 语句中的参数用问号 (?) 来表示。

【例 14.3】 使用 `PreparedStatement` 对象对 xscj 数据库中的 xs 表中添加记录。

TestPrepareStatement.java

```

package org.jdbc;
import java.sql.*;

public class TestPrepareStatement {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        PreparedStatement pstmt = null;
        try {
            /* 加载并注册 MySQL 的 JDBC 驱动 */
            Class.forName("com.mysql.jdbc.Driver");
            /* 建立到 MySQL 数据库的连接 */
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            /* 访问数据库，执行 SQL 语句 */
            stmt = conn.createStatement();
            pstmt = conn.prepareStatement("insert into xs values (?, ?, ?)");
            /* 添加第一条记录 */
            pstmt.setInt(1, 100003);
            pstmt.setString(2, "夏卫国");
            pstmt.setString(3, "基础物理");
            pstmt.executeUpdate();
            /* 添加第二条记录 */
            pstmt.setInt(1, 100004);
            pstmt.setString(2, "李方方");
            pstmt.setString(3, "通信工程");

```

```

        pstmt.executeUpdate();
        /* 添加第三条记录 */
        pstmt.setInt(1, 100005);
        pstmt.setString(2, "刘燕敏");
        pstmt.setString(3, "生物学");
        pstmt.executeUpdate();
        rs = stmt.executeQuery("select * from xs");
        while(rs.next()) {
            System.out.println(rs.getInt("id"));
            System.out.println(rs.getString("name"));
            System.out.println(rs.getString("profession"));
        }
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            if(rs != null) {
                rs.close();                // 关闭 ResultSet 对象
                rs = null;
            }
            if (pstmt != null) {
                pstmt.close();            // 关闭 PreparedStatement 对象
                pstmt = null;
            }
            if(stmt != null) {
                stmt.close();            // 关闭 Statement 对象
                stmt = null;
            }
            if (conn != null) {
                conn.close();            // 关闭 Connection 对象
                conn = null;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

程序运行结果：

```
100001
王军
计算机
100003
夏卫国
基础物理
100004
李方方
通信工程
100005
刘燕敬
生物学
```

14.6 批处理

有时，在程序中需要向数据库中插入、更新或删除大批量的数据，如果只使用 **Statement** 接口中的 `execute()` 方法或 `executeUpdate()` 方法依次提交更新，则既低效又不方便。为解决问题，从 **JDBC 2.0** 开始，允许执行批量更新操作，它能显著提高操作数据库的效率。在 **Statement** 接口中提供了批量更新操作的方法。

- `void addBatch(String sql) throws SQLException`

功能：将给定的 **SQL** 命令添加到此 **Statement** 对象的当前命令列表中。

- `int[] executeBatch() throws SQLException`

功能：将一批命令提交给数据库来执行，如果全部命令执行成功，则返回更新计数组成的数组。返回数组的 `int` 元素的排序对应于批中的命令，批中的命令根据被添加到批中的顺序排序。

【例 14.4】 运用批量更新操作向数据库 `xscj` 中的 `xs` 表中添加一批数据。

TestBatch.java

```
package org.jdbc;
import java.sql.*;
public class TestBatch {
    public static void main(String[] args) {
        ResultSet rs = null;
        Connection conn = null;
        Statement stmt = null;
        PreparedStatement pstmt = null;
        try {
            /* 加载并注册 MySQL 的 JDBC 驱动 */
            Class.forName("com.mysql.jdbc.Driver");
            /* 建立到 MySQL 数据库的连接 */
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            /* 访问数据库，执行 SQL 语句 */
            pstmt = conn.prepareStatement("insert into xs values (?, ?, ?)");
            pstmt.setInt(1, 100011);
            pstmt.setString(2, "马丽丽");
            pstmt.setString(3, "计算机");
            pstmt.addBatch();
            pstmt.setInt(1, 100012);
            pstmt.setString(2, "王玉");
```



```

        pstmt.setString(3, "心理学");
        pstmt.addBatch(); // 加入一条 SQL 语句
        pstmt.setInt(1, 100013);
        pstmt.setString(2, "孙研");
        pstmt.setString(3, "材料科学");
        pstmt.addBatch();
        pstmt.executeBatch(); // 批量更新
        stmt = conn.createStatement();
        rs = stmt.executeQuery("select * from xs");
        while(rs.next()) {
            System.out.print(rs.getInt("id"));
            System.out.print(rs.getString("name"));
            System.out.print(rs.getString("profession"));
            System.out.println();
        }
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            if(rs != null) {
                rs.close(); // 关闭 ResultSet 对象
                rs = null;
            }
            if (pstmt != null) {
                pstmt.close(); // 关闭 PreparedStatement 对象
                pstmt = null;
            }
            if(stmt != null) {
                stmt.close(); // 关闭 Statement 对象
                stmt = null;
            }
            if (conn != null) {
                conn.close(); // 关闭 Connection 对象
                conn = null;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

程序运行结果：

```
100001王军计算机
100003夏卫国基础物理
100004李方方通信工程
100005刘燕敏生物学
100011马丽丽计算机
100012王玉心理学
100013孙研材料科学
```

14.7 事务处理

事务处理是由单一的逻辑单位完成的一系列操作，它由一系列对数据库的操作组成。事务处

理在数据库系统中主要用来实现数据完整性，所有遵守 JDBC 规范的 JDBC 驱动程序都支持事务处理。当在一个事务中执行多个操作时，只有事务中的所有操作成功，才意味着整个事务成功。只要有一个操作失败，整个事务就失败，该事务会回滚（rollback）到最初的状态。

当一个连接对象被创建时，默认情况下事务被设置为自动提交状态。这意味着每次执行一条 SQL 语句时，如果执行成功，就会自动调用 commit()方法向数据库提交，也就不能再回滚了。为了将多条 SQL 语句作为一个事务执行，可以设置 Connection 对象的 setAutoCommit(false)。然后在所有的 SQL 语句成功执行后，显式调用 Connection 对象的 commit()方法来提交事务，或者在执行出错时调用 Connection 对象的 rollback()方法来回滚事务。

【例 14.5】 使用事务处理功能并使用了 JDBC 的批处理向 xscj 数据库的 xs 表中添加数据。

TestTransaction.java

```
package org.jdbc;
import java.sql.*;
public class TestTransaction {
    public static void main(String[] args) {
        ResultSet rs = null;
        Connection conn = null;
        Statement stmt = null;
        try {
            /* 加载并注册 MySQL 的 JDBC 驱动 */
            Class.forName("com.mysql.jdbc.Driver");
            /* 建立到 MySQL 数据库的连接 */
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            conn.setAutoCommit(false); // 禁用自动提交
            /* 访问数据库，执行 SQL 语句 */
            stmt = conn.createStatement();
            stmt.addBatch("insert into xs values (100006, '李明', '计算机');");
            stmt.addBatch("insert into xs values (100007, '赵琳', '通信工程');");
            stmt.addBatch("insert into xs values (100008, '吴伟', '临床医学');");
            stmt.addBatch("insert into xs values (100009, '张薇', '生物化学');");
            stmt.addBatch("insert into xs values (100010, '罗峰', '机械设计');");
            stmt.executeBatch(); // 提交一批命令
            conn.commit();
            conn.setAutoCommit(true); // 开启自动提交
            stmt = conn.createStatement();
```

```

rs = stmt.executeQuery("select * from xs");
while(rs.next()) {
    System.out.print(rs.getInt("id"));
    System.out.print(rs.getString("name"));
    System.out.print(rs.getString("profession"));
    System.out.println();
}
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
    try {
        if (conn != null) {
            conn.rollback();           // 回滚事务
            conn.setAutoCommit(true); // 开启自动提交
        }
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
} finally {
    try {
        if(rs != null) {
            rs.close();           // 关闭 ResultSet 对象
            rs = null;
        }
        if(stmt != null) {
            stmt.close();         // 关闭 Statement 对象
            stmt = null;
        }
        if(conn != null) {
            conn.close();         // 关闭 Connection 对象
            conn = null;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}

```

程序运行结果:

```
100001王军计算机
100003夏卫国基础物理
100004李方方通信工程
100005刘燕敏生物学
100006李明计算机
100007赵琳通信工程
100008吴伟临床医学
100009张薇生物化学
100010罗峰机械设计
100011马丽丽计算机
100012王玉心理学
100013孙研材料科学
```

14.8 综合实例

在这个综合实例中，使用另一个数据库 `company`。在前面的示例中，所用的数据库 `xscj` 和 `xs` 表是在 MySQL Command Line Client 命令行中输入 SQL 语句创建的。这里改在 Java 程序中创建数据库和表，综合运用 JDBC 编程的批处理和事务处理。

UseJDBC.java

```
package org.jdbc;
import java.sql.*;
public class UseJDBC {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        PreparedStatement pstmt = null;
        try {
            /* 加载并注册 MySQL 的 JDBC 驱动 */
            Class.forName("com.mysql.jdbc.Driver");
            /* 建立到 MySQL 数据库的 URL */
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/mysql", "root", "123456");
            /* 访问数据库，执行 SQL 语句 */
            stmt = conn.createStatement();
            stmt.executeUpdate("create database company"); // 创建数据库
            stmt.executeUpdate("use company"); // 使 company 成为当前数据库
            /* 创建表结构 */
            stmt.executeUpdate("create table employee(id int(6) not null primary key
                ,name char(10) not null,salary float(10))");
            /* 加入 sql 命令到命令列表中 */
            stmt.addBatch("insert into employee values (200001, '王林',3000)");
            stmt.addBatch("insert into employee values (200002, '程明',3200)");
            stmt.addBatch("insert into employee values (200003, '李红庆',4000)");
            stmt.addBatch("insert into employee values (200004, '王敏',2500)");
            stmt.addBatch("insert into employee values (200005, '林强',2800)");
            stmt.executeBatch(); // 批量处理
            conn.commit(); // 提交事务
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

rs = stmt.executeQuery("select * from employee");
while(rs.next()) {
    System.out.print(rs.getInt("id"));
    System.out.print(rs.getString("name"));
    System.out.print(rs.getString("salary"));
    System.out.println();
}
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
    try {
        if (conn != null) {
            conn.rollback();           // 事务回滚
            conn.setAutoCommit(true);  // 开启自动提交
        }
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
} finally {
    try {
        if(rs != null) {
            rs.close();               // 关闭 ResultSet 对象
            rs = null;
        }
        if(stmt != null) {
            stmt.close();             // 关闭 Statement 对象
            stmt = null;
        }
        if(conn != null) {
            conn.close();             // 关闭 Connection 对象
            conn = null;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}

```

程序运行结果:

```

200001王林3000
200002程明3200
200003李红庆4000
200004王敏2500
200005林强2800

```

第 2 部分 习 题 集

第 1 章 Java和Eclipse集成开发环境

1. Java 技术体系主要由哪三个部分组成？它们分别应用于什么场合？
2. 简述 Java 语言的主要特点。
3. `main()`方法在 Java 程序中起何种作用？能否从一个类中调用另一个类的 `main()`方法？
4. 在 JDK 环境下，用什么命令编译 Java 程序？用什么命令执行 Java 程序？
5. 用 Eclipse 开发工具设计一个 Java 程序，在控制台上打印“Hello,World!”。

第 2 章 Java语法基础

1. 以下哪些是合法的标识符？
工资 `_123` `4foots` `$123` `12a_2xy` `exam-1` `x+y`
2. Java 中各个基本数据类型分别占多少个字节？Java 中存在无符号数吗？
3. 将下列代数式写成 Java 表达式。
(1) ax^2+bx+c (2) $(x+y)^3$ (3) $(a+b)^2/(a-b)$
4. 计算下列表达式的值。
(1) `x+y%4*(int)(x+z)%3/2` 其中：`x=3.5`, `y=13`, `z=2.5`
(2) `(int)x%(int)y+(double)(z*w)` 其中：`x=2.5`, `y=4.5`, `z=4`, `w=3`
5. 设 `int a=10`，则下列表达式运算后，`a` 的值是多少？
(1) `a += a -= a *= a /= a` (2) `a % (7%2)`
(3) `a /= a+a` (4) `a >>= 32`
6. 一个数如果恰好等于它的因子之和，这个数就称为“完数”。例如：6 的因子为 1、2、3，而 $6=1+2+3$ ，因此 6 是“完数”。编写程序找出 1 000 以内的所有“完数”。
7. 将 `int n` 的值反序打印。例如，`n=1234`，则打印出“4321”。
8. 计算 $1!+2!+\dots+n!$ ，变量 `int n` 的初始值在程序中指定。

第 3 章 Java面向对象编程 (上)

1. 类的成员初始化的顺序如何？静态数据的初始化和非静态实例初始化又是如何？
2. 简述 `static` 数据成员与非 `static` 数据成员的主要区别。
3. 简述 `this` 引用的作用。
4. 创建一个类，它包含一个未初始化的 `String` 引用。验证该引用被 Java 初始化成 `null`。
5. 创建一个带默认的构造方法（无参构造方法）的类，在构造方法中打印一条消息。为这个类创建一个对象。

第 4 章 Java面向对象编程 (下)

1. 什么是接口？什么是抽象类？
2. 简述 Java 继承的主要内容。
3. 简述 Java 运行时多态性的含义。
4. 创建一个不包含任何抽象方法的抽象类，从它那里导出一个子类，并添加一个方法。
创建一个静态方法，它可以接受父类的引用，将其向下转型到子类中，然后再调用该静态方法。
5. 定义两个包，在一个包中创建一个接口，内含三个方法，在另一个包中实现此接口。

第 5 章 常用类与异常处理

1. 什么是异常？
2. 简述 try-catch-finally 语句的执行过程。
3. throws 子句与 throw 语句的区别与联系。
4. 什么样的对象才能由 throw 语句抛出？
5. 在程序中，什么情况下使用 if 语句来处理程序错误？什么情况下用 try-catch-finally 语句作为异常处理？
6. 设计一个 Java 程序，判断一个字符串是否是回文。回文指字符串从左向右读与从右向左读是一样的。

第 6 章 数组与枚举

1. 一个 4 位数的 9 倍是该 4 位数的反序数。例如：4 位数 1089 的 9 倍是 9801。9801 是 1089 的反序数。设计一个 Java 程序，打印出所有具有这种特性的 4 位数。
2. 设计一个 Java 程序，从低到高将从命令行中读取的一组数字进行升序排列。
3. 编程求两个 3 阶矩阵的相加。
4. 创建一个 enum，它包含纸币中最小面值的 3 种类型，通过 values() 循环并打印每一个值及其 ordinal()。
5. 为上面的例子的 enum 编写一个 switch 语句，对于每一个 case，输出该特定货币的描述。

第 7 章 容器和泛型

1. 简述 Collection 和 Collections 的区别。
2. Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢？是用 “= =” 还是 equals()？它们有何区别？
3. List、Set、Map 是否继承自 Collection 接口？
4. 两个对象值相同 (x.equals(y) == true)，但却有不同的 hashCode，这句话对不对？
5. 为什么在创建容器类时使用泛型？使用泛型带来哪些好处？
6. 写一个方法，使用 Iterator 遍历 Collection，并打印容器中每个对象的 toString()。填充

各种类型的 Collection，然后对其使用此方法。

7. 用键值对填充一个 HashMap，打印结果，通过散列码来展示其排序。抽取键值对，按照键进行排序。

第 8 章 Java输入/输出系统

1. 什么是对象的序列化？简述在 Java 程序中进行对象序列化的步骤。
2. 如何在 Java 程序中定制对象的序列化？
3. 使用 FileReader 和 FileWriter 类实现文件的复制。

第 9 章 AWT组件及应用

1. 请说明 FlowLayout、BorderLayout、GridLayout、CardLayout、Null 这几种布局管理器的特点。
2. 组件与容器的区别是什么？
3. 在 AWT 组件中，有几种实现监听接口的方式？
4. 选择题：

(1) 在 Java 语言中，用来构建 GUI 的工具可以分为_____和_____。

(a) 按钮 (b) 组件 (c) 窗体 (d) 容器

(2) 下面不属于“组件”的是_____。

(a) 列表框 (b) 窗体 (c) 菜单

(3) 下面不属于“容器”的是_____。

(a) 多行文本编辑框 (b) 对话框 (c) 窗体

(4) 容器可以被添加到其他容器中去。_____

(a) 正确 (b) 不正确

5. 以下哪些布局管理器会保持组件的最佳宽度和高度？_____

(a) BorderLayout (b) BoxLayout (c) FlowLayout

(d) GridLayout (e) GridBagLayout

6. 结合使用 TextField 和 TextArea。窗口上有一个文本框 TextField 和一个文本区 TextArea，在单行编辑框中输入文字，按回车键将把输入的文字添加到多行编辑框中。

7. 采用 GridLayout 布局管理器，在容器中添加 6 个 Button，单击任意一个 Button，网格的划分方式都会发生变化。

第 10 章 Swing组件及应用

1. 选择题

(1) 组件_____适合于提供密码输入界面。

(a) JTextArea (b) JTextField (c) JPasswordField

(2) 在多行文本编辑框中_____输入超过程序中定义的行数。

(a) 能 (b) 不能

(3) 可以使用_____来清除单行或多行文本框 txt 中的文本。

(a) txt.clearText() (b) txt.setText("")

- (c) txt.deleteText() (d) 以上都可以
- (4) 用_____来获得口令输入框 pwd 中的文本。
- (a) pwd.getText() (b) pwd.getPassword()
- 2. Swing 组件与普通的 AWT 组件有什么区别?
- 3. 编写程序使用 List 在两个列表框之间交换元素。
- 4. 建立一个 JFrame 对象和 JPanel 对象, 注意和 AWT 中 Frame 及 Panel 相比较。

第 11 章 并 发

- 1. 简述程序、进程、线程之间的区别与联系。
- 2. 创建线程有几种方法? 简述每一种方法。
- 3. 线程有几种状态? 简述线程状态之间的转换过程。

第 13 章 Java网络编程

- 1. 简述 TCP 和 UDP 通信的各自特点。什么情况下使用 TCP 通信? 什么情况下使用 UDP 通信?
- 2. 为什么 TCP 通常情况下要使用多线程方式?
- 3. UDP 服务器是采用多线程方式好一些还是采用循环方式好一些?
- 4. 简述组播的基本内容。
- 5. 设计一个程序, 返回请求域名的相关联的 IP 地址, 并返回本地主机的 IP 地址。

第 14 章 JDBC编程

- 1. JDBC 的驱动程序可分为哪几种类型?
 - 2. 哪几类 JDBC 驱动程序完全用 Java 语言编写? _____
 - (a) 第一类 (b) 第二类 (c) 第三类 (d) 第四类
 - 3. 以下哪些对象由 Connection 创建? _____
 - (a) Statement 对象 (b) PreparedStatement 对象 (c) ResultSet 对象
 - 4. 以下哪些方法属于 PreparedStatement 接口的方法? _____
 - (a) addBatch() (b) connect() (c) execute() (d) first()
 - 5. 以下哪些方法属于 ResultSet 接口的方法? _____
 - (a) addBatch() (b) next() (c) getInt() (d) getResultSet()
 - 6. 在 MySQL 命令行中创建 xscj 数据库, 并在此数据库中创建一个 student 表, 表中有 5 列:
- | | | | |
|---------|---------|----|---------------------|
| id | int | 4 | // 学号 |
| name | char | 16 | // 姓名 |
| sex | tinyint | 1 | // 性别: 1 表示男, 0 表示女 |
| math | integer | | // 数学成绩 |
| english | integer | | // 英语成绩 |

请编写程序, 实现对 student 表的查询、插入、删除、修改功能。

第 3 部分 实 验

计算机学科是一门实践性很强的学科。计算机应用能力的培养和提高，要靠大量的上机实验。为了配合本书的学习，这里汇集了 Java 的上机实验题，可以在前面各章节的学习过程中选择相应的实验题进行上机操作，加深对 Java 的理解和培养实际编程能力。本书的实验练习将在 Eclipse 环境下调试、编译、运行。

实验 1 Java和Eclipse集成开发环境

实验目的

- (1) 熟悉 Eclipse 的集成开发环境。
- (2) 建立 Eclipse 的工程和应用。
- (3) 在 Eclipse 中编辑、调试、运行 Java 程序。

实验准备

- (1) 复习本书第 1 章中有关内容。
- (2) 安装 Eclipse。

实验内容

创建 Java 项目，熟练掌握 Eclipse 的集成开发环境。创建一个 Java 项目，并编辑、调试、运行 Java 程序，需要以下三个步骤。

1. 创建Java项目

进入图 T1.1，在工作台主窗口中，依次选择“File”→“New”→“Java Project”命令，打开新建项目向导，在“Project name”栏中输入项目名“MyProject_01”，其他选项默认，单击“Finish”按钮，项目创建成功，项目“MyProject_01”将出现在左边的 Navigator（导航器）中。

2. 创建Java包

在 Navigator 中右击项目“MyProject_01”，选择“New”→“Package”命令，如图 T1.2 所示，在“Name”栏中输入包名“org.exercise”，单击“Finish”按钮完成包的创建。



图 T1.1 创建 Java 项目

3. 创建Java类

右击项目“MyProject_01”的包“org.exercise”，选择“New”→“Class”命令，如图 T1.3 所示，在“Name”栏中输入类名“HelloWorld”，单击“Finish”按钮完成类的创建。这时可以编写 Java 程序了。双击“HelloWorld.java”，输入“HelloWorld.java”源程序，如图 T1.4 所示，单击“保存”按钮。右击“HelloWorld.java”，选择“Run as”→“Java Application”，将在控制台上打印以下字符串。

```
hello world  11
```

整个程序流程到此结束。

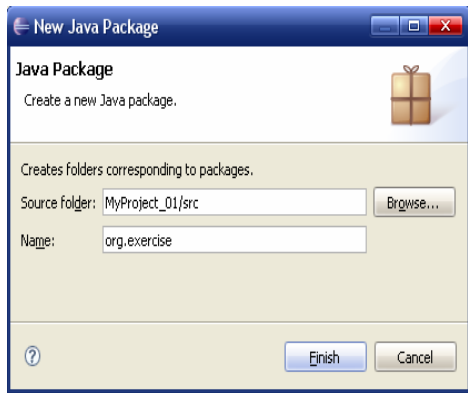


图 T1.2 创建 Java 包

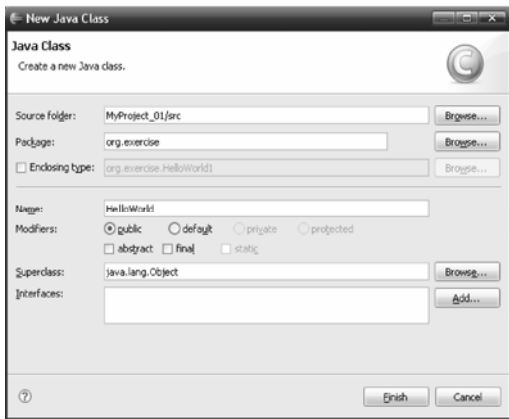


图 T1.3 创建 Java 类

Eclipse 强大的集成开发环境提供了编辑、调试和运行 Java 程序的便利性。比如，在上述的 HelloWorld.java 程序中，如果想更改变量的名字就很方便。若想把成员变量 x 的名字改为 a，只要在变量 x 上右击，在弹出的快捷菜单中选择“Refactor”，并在“Refactor”的下级菜单中选择“Rename”，输入自己想要定义的名字，这样，变量“x”就完成重新命名。依照上述办法，也可以把变量“s”命名为“words”。不过，类名和包名的重新命名需要在左边的导航器中进行。方式和上面的一样。

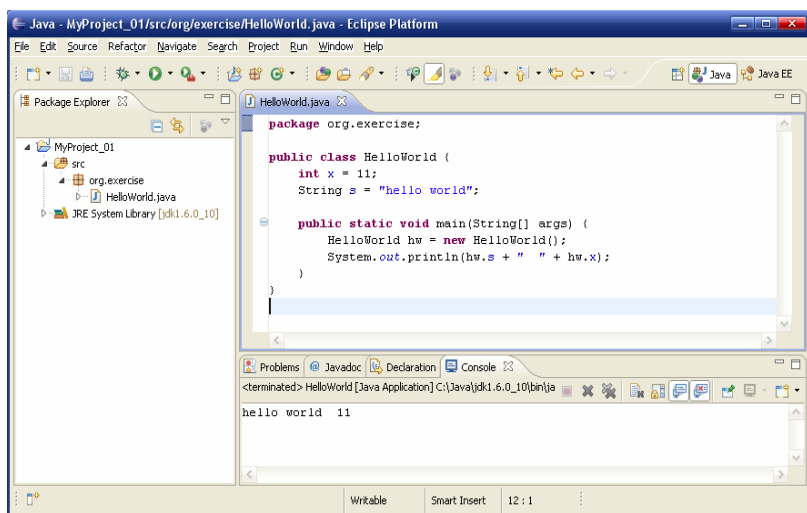


图 T1.4 Eclipse 工作台

思考与练习题

- (1) 如何修改工程属性？
- (2) 在代码中制造一些错误，如将 `String` 改为 `string`，将分号改为其他标点符号如逗号等，运行程序，观看效果。
- (3) 如何格式化 Java 代码？
- (4) 依照本书中的第 1 章程序，分别求半径为 2mm、3.7mm、5.12mm 的圆面积。

实验 2 Java语法基础

实验目的

- (1) 掌握 Java 基本语句的使用。
- (2) 理解递归的原理。

实验准备

- (1) 复习本书第 2 章中有关内容。
- (2) 打开 Eclipse 集成开发环境。

实验内容

【实验 2.1】 求 a 到 b 之间的所有质数，每行显示 c 个。 a 、 b 、 c 的值通过命令行传递。

质数也称素数，指的是只能被 1 和它自身整除的正整数。

GetPrime.java

```
public class GetPrime {
    public static void main(String args[]) {
        int a, b, c;
        // 读取命令行中的参数并把字符串类型转换为整型
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
        c = Integer.parseInt(args[2]);
        boolean flag;
        int m, p, count = 0;
        for (m = a; m <= b; m++) {
            flag = true;
            for (p = 2; p <= m / 2; p++)
                if (m % p == 0) {
                    flag = false;
                    break;
                }
            if (flag) {
                System.out.print(m + "\t");
                count++;
                if (count % c == 0) // 每行中只输出 c 个素数
                    System.out.println();
            }
        }
    }
}
```

右击 “GetPrime.java”，选择 “Run As” → “Run Configurations”，如图 T2.1 所示，选择 “Main” 标签页，在 “Project” 栏中选择 “MyProject_01”，在 “Main class” 栏中选择 “GetPrime”，选择 “Arguments” 标签页，在 “Program arguments” 栏中输入 “10 500 8”，然后单击 “Run” 按钮，运行程序。

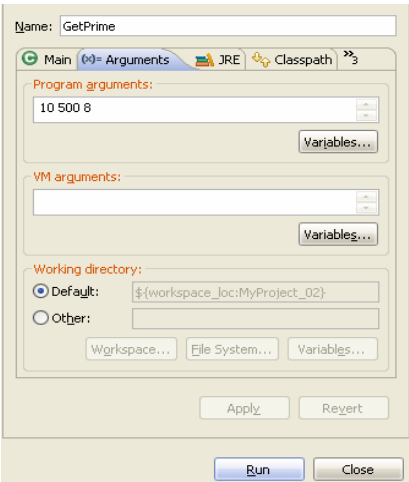


图 T2.1 生成所有质数

程序运行结果:

11	13	17	19	23	29	31	37
41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107
109	113	127	131	137	139	149	151
157	163	167	173	179	181	191	193
197	199	211	223	227	229	233	239
241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337
347	349	353	359	367	373	379	383
389	397	401	409	419	421	431	433
439	443	449	457	461	463	467	479
487	491	499					

【实验 2.2】 输入两个整数 m 和 n ，求其最大公约数和最小公倍数。

CommonMultiply.java

```
public class CommonMultiply {
    public static void main(String[] args) {
        int m, n, r, gcd, lcm = 0;
        m = Integer.parseInt(args[0]);           // 把字符串转换为整型
        n = Integer.parseInt(args[1]);
        lcm = m * n;
        while ((r = m % n) != 0) {
            m = n;
            n = r;
        }
        gcd = n;
        lcm = lcm / gcd;
        System.out.println("最大公约数:"+gcd);    // 打印出最大公约数
        System.out.println("最小公倍数:"+lcm);    // 打印出最小公倍数
    }
}
```

右击“CommonMultiply.java”，选择“Run As”→“Run Configurations”，如图 T2.2 所示，选择“Main”标签页，在“Project”栏中选择“MyProject_02”，在“Main class”栏中选择“CommonMultiply”，选择“Arguments”标签页，在“Program arguments”栏中输入“9 15”，然后单击“Run”按钮，运行程序。

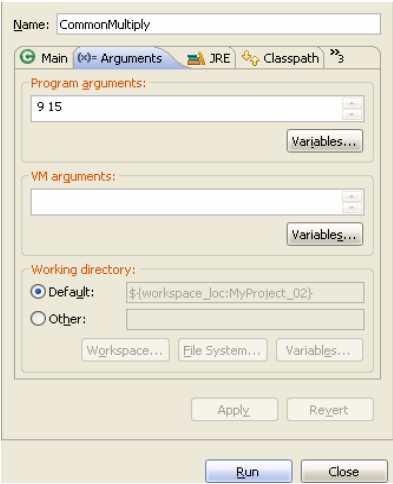


图 T2.2 求最大公约数和最小公倍数

程序运行结果:

```
最大公约数:3  
最小公倍数:45
```

【实验 2.3】 设计一个递归程序, 计算 $n!$ 。

分析: $n = n * (n-1)!$, 当 $n = 0$ 时, $0! = 1$ 。

Factorial.java

```
public class Factorial {  
    static int F(int n) {  
        return n == 0 ? 1 : n * F(n - 1);  
    }  
    public static void main(String[] args) {  
        System.out.println("10!是: " + F(10));  
    }  
}
```

程序运行结果:

```
10!是: 3628800
```

思考与练习题

- (1) 求 2~400 之间的所有质数, 每行显示 5 个。
- (2) 求 $1!+2!+3!+4!+5!$ 的和。
- (3) 设计一个 Java 的递归程序, 打印出 $2+4+6+\cdots+100$ 之和。
- (4) 一个数如果恰好等于它的因子之和, 这个数就称为“完数”。例如, 6 的因子为 1、2、3, 而 $6 = 1+2+3$, 因此 6 是“完数”。编写程序找出 1 000 以内的所有“完数”。

实验 3 Java面向对象编程 (上)

实验目的

- (1) 掌握面向对象的思想, 用面向对象的思维编写 Java 程序。
- (2) 掌握方法的重载和理解 static 关键字。

实验准备

- (1) 复习本书第 3 章的有关内容。
- (2) 打开 Eclipse 集成开发环境。

实验内容

【实验 3.1】 根据所传递参数的数据类型调用适当的 max()方法。

TestOverLoad.java

```
public class TestOverLoad {
    void max(int a , int b) {
        System.out.print("int: ");
        System.out.println( a > b ? a : b );
    }
    void max(short a , short b) {
        System.out.print("short: ");
        System.out.println( a > b ? a : b );
    }
    void max(double a, double b) {
        System.out.print("double: ");
        System.out.println( a > b ? a : b );
    }
    public static void main(String[] args) {
        TestOverLoad t = new TestOverLoad();
        t.max(300, 400);
        short a = 100;
        short b = 110;
        t.max(a, b);
        t.max(300.5, 400.5);
    }
}
```

程序运行结果:

```
int: 400
short: 110
double: 400.5
```

【实验 3.2】 使用 static 关键字修饰类的成员和代码块。

StaticDemo.java

```
class StaticDemo {
    static int m = 10;                // 静态变量
    static int n;
    static void method(int a){        // 静态方法
        System.out.println("a = " + a);
        System.out.println("m = " + m);
        System.out.println("n = " + n);
    }
    static {                          // 静态代码块
        System.out.println("static block is initialized.");
    }
}
```



```

        n = m * 4;
    }
    public static void main(String args[]) {
        method(42);
    }
}

```

程序运行结果：

```

static block is initialized.
a = 42
m = 10
n = 40

```

【实验 3.3】 编写具有两个（重载）构造方法的类，并在第一个构造方法中通过 `this` 调用第二个构造方法。

OverloadedConstructors.java

```

package org.overload;
class Teacher {
    Teacher(int i) {
        this("doctor"); // 在构造方法中位于第一条语句
        int yearsTraining = i;
        System.out.println("The teacher teaches " + i + " years ");
    }
    Teacher(String s) {
        String degree = s;
        System.out.println("The teacher's degree is " + s );
    }
    void teach() {
        System.out.println("teacher teaches very good!");
    }
}
public class OverloadedConstructors {
    public static void main(String[] args) {
        new Teacher(8).teach();
    }
}

```

程序运行结果：

```

The teacher's degree is doctor
The teacher teaches 8 years
teacher teaches very good!

```

思考与练习题

- (1) 为什么 `main()` 方法是静态的？
- (2) 创建一个重载构造器的类，令其接收一个字符串参数，并在构造器中把自己的信息和接收的参数打印出来。
- (3) 创建 `Person` 类，分别用三种构造方法创建三个 `Person` 对象。

实验 4 Java面向对象编程（下）

实验目的

- (1) 掌握 Java 的接口、继承和多态。
- (2) 掌握类的初始化顺序。

实验准备

- (1) 复习第 4 章的内容，掌握 Java 的继承和多态。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 4.1】 一个子类继承父类，运行程序可以观察父类与子类的初始化顺序。

Cartoon.java

```
class Art {  
    Art() {  
        System.out.println("Art constructor");  
    }  
}  
  
class Drawing extends Art {  
    Drawing() {  
        System.out.println("Drawing constructor");  
    }  
}  
  
public class Cartoon extends Drawing {  
    public Cartoon() {  
        System.out.println("Cartoon constructor");  
    }  
    public static void main(String[] args) {  
        Cartoon cart = new Cartoon();  
    }  
}
```

程序运行结果：

```
Art constructor  
Drawing constructor  
Cartoon constructor
```

【实验 4.2】 综合运用继承与多态的知识。

```

package org.extend;
class A {
    public void method1() {
        System.out.println("invoke A method1");
    }
    public void method2() {
        method1();
    }
}
class B extends A {
    public void method3() {
        System.out.println("invoke B method3");
    }
}
class C {
    public static void main(String[] args) {
        B b = new B();
        A a = b;                                // 向上类型转换
        callA(a);
        callA(new B());
    }
    public static void callA(A a) {
        B b = (B) a;                            // 强制类型转换
        b.method1();
        b.method2();
        b.method3();
    }
}

```

程序运行结果:

```

invoke A method1
invoke A method1
invoke B method3
invoke A method1
invoke A method1
invoke B method3

```

说明：从程序的运行结果可以看出，程序能够自动将类 B 的实例对象 b 直接赋值给 A 类的引用类型变量，即子类能够自动转换成父类型，也就是发生了向上类型转换。当想把一个父类型的对象赋值给子类型时，必须进行强制类型转换。

【实验 4.3】 创建一个带 final 方法的类，由此继承产生一个类并重写该方法。

FinalOverrideEx.java

```

package org.extend;
class FinalMethod {
    final void f() {                            // final 方法
        System.out.println("FinalMethod.f()");
    }
}

```

```

    void g() {
        System.out.println("FinalMethod.g()");
    }
    final void h() {
        System.out.println("FinalMethod.h()");
    }
}
class OverrideFinal extends FinalMethod {
    public void g() {                // 重写父类的 g()方法
        System.out.println("OverrideFinal.g()");
    }
}
public class FinalOverrideEx {
    public static void main(String[] args) {
        OverrideFinal of = new OverrideFinal();
        of.f(); of.g(); of.h();
        FinalMethod wf = of;
        wf.f(); wf.g(); wf.h();
    }
}

```

程序运行结果:

```

FinalMethod.f()
OverrideFinal.g()
FinalMethod.h()
FinalMethod.f()
OverrideFinal.g()
FinalMethod.h()

```

思考与练习题

- (1) 编写一个程序，使其能够展示父类与子类的初始化顺序，并向其父类和子类中添加成员对象，说明构建期间初始化发生的顺序。
- (2) 创建一个包含两个方法的父类。在第一个方法中可以调用第二个方法。再产生一个继承该父类的子类，且覆盖父类的第二个方法。为该子类创建一个对象，将它向上转型到父类型并调用第一个方法。
- (3) 编写一个程序，分别创建一个 `static final` 变量和一个 `final` 类。

实验 5 常用类与异常处理

实验目的

- (1) 掌握 Java 常用类的使用。
- (2) 理解 Java 的异常处理。

实验准备

- (1) 进入 Eclipse 集成开发环境。
- (2) 复习第 5 章的内容，掌握 Java 常用类的使用和正则表达式，理解异常的抛出、捕获与处理。

实验内容

【实验 5.1】 统计一个字符串中单词的个数。设单词之间用一个或多个空格分隔，该字符串只由字母与空格组成。

TestString.java

```
import java.util.*;
public class TestString {
    public static void main(String[] args) {
        String data = "This is a String";
        StringTokenizer st = new StringTokenizer(data);
        int count = st.countTokens();           // 计算单词总数
        System.out.println("原串是: " + data);
        System.out.println("各个单词如下: ");
        while (st.hasMoreTokens()) {           // 还有子串时
            String s = st.nextToken();         // 取出下一个子串
            System.out.println(s);
        }
        System.out.println("单词总数: " + count);
    }
}
```

程序运行结果：

```
原串是: This is a String
各个单词如下:
This
is
a
String
单词总数: 4
```

【实验 5.2】 使用正则表达式获取 D 盘 test.txt 文本文件中的邮箱地址。

EmailSpider.java

```
import java.io.BufferedReader;
import java.io.*;
import java.util.regex.*;
public class EmailSpider {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("d:\\test.txt"));
            String line = "";
            while((line=br.readLine()) != null) {
```

```

        parse(line);
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}

private static void parse(String line) {
    Pattern p = Pattern.compile("[\\w[.]]+@[\\w[.]]+\\.([\\w]+)"); // 编译正则表达式
    Matcher m = p.matcher(line); // 匹配邮箱地址
    while(m.find()) {
        System.out.println(m.group()); // 打印匹配的邮箱地址
    }
}
}

```

程序运行结果:

```

weiyanhong@163.com
zhangjun@sina.com
lifangfang@sina.com
zhaolin@yahoo.com.cn
chengming@hotmail.com
lingyifan@163.com

```

【实验 5.3】 设计一个异常处理程序，异常类型有除数是 0 异常、空指针异常、数组越界异常等。

ExceptionTest.java

```

public class ExceptionTest {
    public static void main(String args[]) {
        for (int i = 0; i < 3; i++) {
            int k;
            try {
                switch (i) {
                    case 0:
                        int a = 0;
                        k = 10 / a; // 除数为 0 异常
                        break;
                    case 1: // 空指针异常
                        int b[] = null;
                        k = b[0];
                        break;
                    case 2:
                        int c[] = new int[5];
                        k = c[5]; // 数组索引越界异常
                        break;
                }
            } catch (Exception e) {

```

```

        System.out.println("\n 异常" + i + "\n");
        System.out.println(e);
    }
}
}
}
}

```

程序运行结果：

```

异常0
java.lang.ArithmeticException: / by zero
异常1
java.lang.NullPointerException
异常2
java.lang.ArrayIndexOutOfBoundsException: 5

```

思考与练习题

- (1) 设计一个类，统计一个字符串中单词的个数，设单词之间用一个或多个空格分隔，该字符串只由字母与空格组成。
- (2) 编写一个程序，读取一个 Java 源代码文件，打印出代码中的所有的普通字符串。
- (3) 定义一个对象引用并初始化为 `null`，用此引用调用方法。把这调用放在 `try-catch` 子句里以捕获异常。

实验 6 数组与枚举

实验目的

- (1) 熟练运用 Java 的数组。
- (2) 熟悉 Java 的枚举。

实验准备

- (1) 复习本书第 6 章的有关内容。
- (2) 打开 Eclipse 集成开发环境。

实验内容

【实验 6.1】 运用 Java 的二维数组，打印“魔方阵”。所谓魔方阵是指这样的方阵，它的每一行、每一列和对角线之和均相等。要求打印 1~25 之间由自然数构成的魔方阵。

思路分析：

- (1) 第一个位置在第一行的正中。
- (2) 新位置应当处于最近一个插入位置的右上方，但若右上方位置已超出方阵的上边界，则新位置应选列的最下一个位置，若超出右边界，则新位置应选行的最左一个位置。
- (3) 若最近一个插入的元素为 n 的整数倍，则选下面一行同列上的位置为新位置。

Magics.java

```
package org.arrays;
public class Magics {
    public static void main(String[] args) {
        int i = 0;
        int j = 0;
        int m = 0;
        int n = 5;
        j = (n + 1) / 2 - 1;
        int[][] a = new int[n][n];
        a[i][j] = ++m;           // 第一个数在第一行正中
        while (m < n * n) {
            i--;
            j++;
            // 最近插入的元素为 n 的整数倍时，则选下面一行同列上的位置为新位置
            if (m % n == 0 && m > 1) {
                i = i + 2;
                j = j - 1;
            }
            if (i < 0)           // 超出方阵上边界，则新位置应选列的最下一个位置
                i = i + n;
            if (j > (n - 1))     // 超出方阵右边界，则新位置应选行的最左一个位置
                j = j - n;
            a[i][j] = ++m;
        }
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                System.out.print(a[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

程序运行结果：

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

【实验 6.2】 运用 Java 的二维数组，求出两个矩阵相乘的值。在程序中， $a1$ 是一个

2×3 的矩阵， $b1$ 是一个 3×2 的矩阵，求出这两个矩阵的积。

MatrixDemo.java

```
package org.arrays;

public class MatrixDemo {
    public static void main(String[] args) {
        int[][] a1 = { { 1, 2, 3 }, { 4, 5, 6 } };
        int[][] b1 = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
        int[][] c1 = new int[2][2];
        for (int row = 0; row < 2; row++) {
            for (int col = 0; col < 2; col++) {
                c1[row][col] = 0;
                for (int k = 0; k < 3; k++)
                    c1[row][col] += a1[row][k] * b1[k][col];
            }
        }
        for (int row = 0; row < 2; row++) {
            for (int col = 0; col < 2; col++)
                System.out.print( c1[row][col] + "\t");
            System.out.println();
        }
    }
}
```

程序运行结果：

22	28
49	64

【实验 6.3】 设计一个枚举的程序，使用枚举类型常用的 `values()`方法和 `ordinal()`方法。

EnumClass.java

```
package org.enums;

enum Season{
    SPRING, SUMMER, AUTUMN, WINTER
}

public class EnumClass {
    public static void main(String[] args) {
        for (Season s : Season.values()) {
            // 遍历 enum 实例
            //得到 enum 实例在声明的顺序
            System.out.print(s + " ordinal: " + s.ordinal());
            System.out.print(s.compareTo(Season.SUMMER) + " ");
            // 比较是否相等
            System.out.print(s.equals(Season.SUMMER) + " ");
            System.out.print(s == Season.SUMMER);
            System.out.print(s.getDeclaringClass());
            System.out.println(s.name());
            System.out.println("-----");
        }
        for (String s : "SPRING SUMMER AUTUMN WINTER".split(" ")) {
            // 根据给定的名字返回相应的实例
        }
    }
}
```

```

        Season season = Enum.valueOf(Season.class, s);
        System.out.println(season);
    }
}
}

```

程序运行结果：

```

SPRING ordinal: 0-1 false falseclass org.enums.SeasonSPRING
-----
SUMMER ordinal: 10 true trueclass org.enums.SeasonSUMMER
-----
AUTUMN ordinal: 21 false falseclass org.enums.SeasonAUTUMN
-----
WINTER ordinal: 32 false falseclass org.enums.SeasonWINTER
-----
SPRING
SUMMER
AUTUMN
WINTER

```

思考与练习题

- (1) 将一个数组中值按逆序重新存放。例如，原来顺序为 15、12、9、6、3，要求改为 3、6、9、12、15。
- (2) 设计一个类，打印出 4×4 矩阵两对角线元素之和。
- (3) 创建一个 enum，它包含纸币中的所有面值，通过 values() 循环并打印每一个值及其 ordinal()。

实验 7 容器和泛型

实验目的

- (1) 掌握 Java 中常用容器的使用方法。
- (2) 理解泛型的作用。

实验准备

- (1) 复习本书第 7 章的有关内容。
- (2) 打开 Eclipse 集成开发环境。

实验内容

【实验 7.1】 设计一个 Java 程序，统计一个大小写混合的字符串中，每一个英文字母的使用频度。要求小写字母全部转换成大写字母。

ConverseUpper.java

```
package org.container;
import java.util.*;
public class ConverseUpper {
    public static void main(String[] args) {
        String s = "afAsdfAsSgdfGdfgDfGsDfg";
        String str = s.toUpperCase();           // 将字符串中的所有字符转换成大写
        char[] num = str.toCharArray();         // 将字符串转换为 char 数组
        int i = num.length-1;
        TreeMap map = new TreeMap();           // 创建一个 TreeMap 对象
        map.put(num[0], 1);
        for (int k=1; k<=i;k++){
            if(map.containsKey(num[k])){        // 如果在容器中已存在该字母，字母数加 1
                Integer j = (Integer) map.get(num[k]);
                map.put(num[k], ++j);
            }
            else map.put(num[k], 1);             // 如果不存在，将该字母加入到容器中
        }
        System.out.println(map);
        List list = Collections.synchronizedList(new ArrayList());
    }
}
```

程序运行结果：

```
{A=3, D=5, F=6, G=5, S=4}
```

【实验 7.2】 创建一个 Set 容器，用同样的数据多次填充 Set，然后验证此 Set 中没有重复的元素。使用 HashSet、TreeSet 做此测试。

SetTest.java

```
package org.container;
import java.util.*;
public class SetTest {
    public static void main(String[] args) {
        HashSet<String> has = new HashSet<String>();
        has.add("Struts");
        has.add("Hibernate");
        has.add("Spring");
        has.add("Struts");           // 向 HashSet 容器中加入重复元素
        Iterator it = has.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

```

    }
    TreeSet<Integer> ts = new TreeSet<Integer>();
    ts.add(3); ts.add(5);
    ts.add(9); ts.add(6);
    ts.add(3); //向 TreeSet 容器中加入重复元素
    Iterator it2 = ts.iterator();
    while(it2.hasNext()){
        System.out.println(it2.next());
    }
}
}

```

程序运行结果:

```

Hibernate
Struts
Spring
3
5
6
9

```

【实验 7.3】 设计一个泛型的示例，创建 Holder2 对象时，指明是什么类型的对象，将其置于尖括号内。

Holder2.java

```

package org.generics;
class Automobile {}
class Holder1 {
    private Automobile a;
    public Holder1(Automobile a)
    { this.a = a; }
    Automobile get()
    { return a; }
}
public class Holder2<T> {
    private T a;
    public Holder2(T a)
    {this.a = a; }
    public void set(T a)
    { this.a = a; }
    public T get()
    {return a;}
    public static void main(String[] args) {
        // 持有 Automobile 类型的对象
        Holder2<Automobile> h2 =new Holder2<Automobile>(new Automobile());
        Automobile a = h2.get();
        // 持有 String 类型的对象
        Holder2<String> h3 = new Holder2<String>(new String("generics"));
        // 持有 Integer 类型的对象
        Holder2<Integer> h4 = new Holder2<Integer>(new Integer(12));
    }
}

```

```

        String str = h3.get();
        int i = h4.get();
        System.out.println(str);
        System.out.println(i);
    }
}

```

程序运行结果：

```

generics
12

```

思考与练习题

- (1) 设计一个类，写入一系列学生学号和姓名到 **HashMap** 容器中，并根据学号查找出对应的学生姓名。
- (2) 创建一个元音字母 **Set**，计数显示输入文件中所有元音字母的数量总和。
- (3) 创建一个 **Holder** 类，使其能够持有具有系统类型的 3 个对象，并提供相应的读写方法访问这些对象，以及一个可以初始化其持有的 3 个对象的构造器。
- (4) 分别创建一个 **ArrayList** 和 **LinkedList**，比较其操作元素的方法的差异。

实验 8 Java 输入/输出系统

实验目的

- (1) 掌握 Java 常用的 I/O 输入流和输出流。
- (2) 理解流的串接。

实验准备

- (1) 复习本书第 8 章的有关内容。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 8.1】 求 $a \sim b$ 之间的所有质数，每行显示 c 个，结果保存到文本文件中，并在文件的最后写入所求质数的个数。 a 、 b 、 c 的值及文件名在程序运行时通过键盘输入。

GetPrimeToFile.java

```

package org.iostream;
import java.io.*;
public class GetPrimeToFile {

```

```

public static void main(String arg[]) throws IOException {
    int a, b, c, count = 0, m, p;
    boolean flag;
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("请输入 a 的值: ");
    a = Integer.parseInt(in.readLine());
    System.out.println("请输入 b 的值: ");
    b = Integer.parseInt(in.readLine());
    System.out.println("请输入 c 的值: ");
    c = Integer.parseInt(in.readLine());
    System.out.println("请输入要保存的文件名: ");
    FileWriter out = new FileWriter(in.readLine());
    for (m = a; m <= b; m++) {
        flag = true;
        for (p = 2; p < m / 2; p++)
            if (m % p == 0) {
                flag = false;
                break;
            }
        if (flag) {
            count++;
            out.write(m + "\t");
            if (count % c == 0)
                out.write("\r\n");
        }
    }
    out.write("\r\n 共有" + count + "个质数");
    out.close();
    System.out.println("已完成");
}
}

```

运行程序，根据提示符输入数值和文件名。

```

请输入a的值:
2
请输入b的值:
500
请输入c的值:
6
请输入要保存的文件名:
prime.txt
已完成

```

保存在 prime.txt 文件中的内容如下。

```

2   3   4   5   7   11
13  17  19  23  29  31
37  41  43  47  53  59
61  67  71  73  79  83
89  97  101 103 107 109
113 127 131 137 139 149
151 157 163 167 173 179

```

```
181 191 193 197 199 211
223 227 229 233 239 241
251 257 263 269 271 277
281 283 293 307 311 313
317 331 337 347 349 353
359 367 373 379 383 389
397 401 409 419 421 431
433 439 443 449 457 461
463 467 479 487 491 499
共有 96 个质数
```

【实验 8.2】 利用文件流和缓冲流把一个文件 t12.txt 中的内容复制到文件 t13.txt 中。

BufferedFileCopy.java

```
package org.iostream;
import java.io.*;

public class BufferedFileCopy {
    public static void main(String[] args) {
        FileInputStream fis = null;
        FileOutputStream fos = null;
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        int c;
        try {
            // 文件输入流
            fis = new FileInputStream("e:/workbench/MyProject_08/src/org/iostream/t12.txt");
            bis = new BufferedInputStream(fis);           // 串接成带缓冲的输入流
            // 文件输出流
            fos = new FileOutputStream("e:/workbench/MyProject_08/t13.txt");
            bos = new BufferedOutputStream(fos);         // 串接成带缓冲的输出流
            while ((c = bis.read()) != -1)
                bos.write(c);
            bos.flush();                                // 刷新流，强制输出
        } catch (FileNotFoundException e1) {
            System.out.println(e1);
        } catch (IOException e2) {
            System.out.println(e2);
        } finally {
            try {
                if (fis != null)
                    fis.close();
                if (fos != null)
                    fos.close();
                if (bis != null)
                    bis.close();
                if (bos != null)
                    bos.close();
            } catch (IOException e3) {
```

```

        System.out.println(e3);
    }
}
}
}
}

```

运行程序，在“e:\workbench\MyProject_08\src\org\iostream”目录下建立 t12.txt 文件，输入 0~20 的数值，运行程序，那么将把 t12.txt 文件中的内容复制到 t13.txt 文件中。

【实验 8.3】 从键盘上输入一个整数值 n ，打印出 $1+2+\dots+n$ 之和。

ComputeSum.java

```

package org.iostream;
import java.io.*;
public class ComputeSum {
    public static void main(String[] args) throws Exception {
        BufferedReader kr = new BufferedReader(new InputStreamReader(System.in));
        // 键盘转换成一个能读取文本行的字符输入流
        String s;
        int n = 0;
        int sum = 0, k = 0;
        System.out.println("请输入一个整数值: ");
        s = kr.readLine();           // 从键盘读取一行
        n = Integer.parseInt(s);     // 转换成整数值
        for (k = 1; k <= n; k++)
            sum = sum + k;
        System.out.println("1 + 2 + ... + " + n + " = " + sum);
    }
}

```

运行程序，在控制台中输入 100，按下回车键。

程序运行结果：

```

请输入一个整数值:
100
1 + 2 + ... + 100 = 5050

```

思考与练习题

- (1) 设计一个类，用来将文本文件中的内容一行一行地在屏幕上显示出来，每次读取一个文本行。
- (2) 设计一个类，用来从键盘上读入任意数量的字符，并写入到文本文件中，在把字符串从这个文件中复制到另一个文件中。
- (3) 统计一个字符串中英文字母和数字的个数。

实验 9 AWT组件及应用

实验目的

- (1) 掌握常用的 AWT 组件。
- (2) 理解 AWT 事件处理原理。

实验准备

- (1) 复习本书第 9 章的有关内容。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 9.1】 计算圆面积。在第一个文本框中输入圆的半径，将此圆面积的值放入到第二个文本框中。

CircleArea.java

```
package org.awt;
import java.awt.*;
import java.awt.event.*;

public class CircleArea extends Frame implements ActionListener{
    static TextField tf1 = new TextField();
    static TextField tf2 = new TextField();
    static Button bt1 = new Button("圆半径");
    static Button bt2 = new Button("圆面积");
    public static void main(String[] args) {
        CircleArea circle = new CircleArea();
        circle.setLayout(null);
        circle.setBackground(Color.cyan);
        circle.setVisible(true);
        circle.setSize(300, 200);
        circle.add(bt1); circle.add(bt2);
        circle.add(tf1); circle.add(tf2);
        bt1.setBackground(Color.cyan);
        bt1.setBounds(new Rectangle(25, 80, 70, 25));
        tf1.setBounds(new Rectangle(140, 80, 70, 25));
        bt2.setBounds(new Rectangle(25, 130, 70, 25));
        tf2.setBounds(new Rectangle(140, 130, 70, 25));
        bt2.setBackground(Color.cyan);
        bt2.addActionListener(circle);           // 注册事件监听器
    }
    public void actionPerformed(ActionEvent e) {
```

```

String s = tf1.getText();           // 获取文本框中的内容
int i = Integer.parseInt(s);
float d = (float)Math.PI*i*i;      // 计算圆面积
String str = String.valueOf(d);
tf2.setText(str);                   // 把圆面积的值放入到文本框中
}
}

```

运行程序，在第一个文本框中输入圆的半径“5”，单击“圆面积”按钮，计算的圆面积的值放入到第二个文本框中，如图 T9.1 所示。

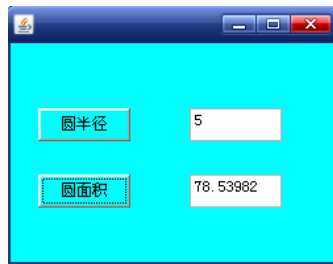


图 T9.1 求圆面积

【实验 9.2】 求 $a \sim b$ 之间的所有质数，每行显示 c 个。 a 、 b 、 c 的值由文本框输入，结果在文本区上显示，并显示质数个数。

TestPrime.java

```

package org.awt;
import java.awt.*;
import java.awt.event.*;
public class TestPrime {
    static TextField tf1 = new TextField();
    static TextField tf2 = new TextField();
    static TextField tf3 = new TextField();
    // 设置有垂直和水平滚动条的文本区
    static TextArea ta = new TextArea("", 5, 10, TextArea.SCROLLBARS_BOTH);
    static TextField tf4 = new TextField();
    static int num = 0;
    public static void main(String[] args) {
        Frame f = new Frame("求 a 到 b 之间的质数");
        f.setBackground(Color.cyan);
        f.setSize(new Dimension(500, 400));
        f.setLayout(null);
        f.setLocation(300, 300);
        Button bt1 = new Button("求 a 到 b 之间的质数");
        bt1.setBackground(Color.cyan);
    }
}

```

```

        bt1.addActionListener(new GetAction());
        Button bt2 = new Button("质数个数");
        bt2.setBackground(Color.cyan);
        Label l1 = new Label("输入 a 的值");
        Label l2 = new Label("输入 b 的值");
        Label l3 = new Label("每行显示个数");
        tf1.setBounds(new Rectangle(40, 70, 70, 25));
        tf2.setBounds(new Rectangle(130, 70, 70, 25));
        tf3.setBounds(new Rectangle(220, 70, 70, 25));
        ta.setEditable(true);
        ta.setText("");
        ta.setBackground(Color.white);
        ta.setBounds(new Rectangle(40, 120, 400, 200));
        l1.setBounds(new Rectangle(40, 40, 60, 25));
        l2.setBounds(new Rectangle(130, 40, 60, 25));
        l3.setBounds(new Rectangle(220, 40, 120, 25));
        bt1.setBounds(new Rectangle(340, 40, 120, 25));
        bt2.setBounds(new Rectangle(40, 350, 50, 25));
        tf4.setBounds(new Rectangle(130, 350, 70, 25));
        f.add(l1); f.add(l2); f.add(l3);
        f.add(bt1); f.add(bt2);
        f.add(ta); f.add(tf1);
        f.add(tf2); f.add(tf3);
        f.add(tf4);
        f.setLayout(null);
        f.setVisible(true);
        f.addWindowListener(new WindowHandler11());           // 注册事件监听器
    }
}
// *****方法 windowClosing 就是当窗口关闭时的处理动作*****
class WindowHandler11 extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(1); // 关闭窗口
    }
}
class GetAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String text1 = TestPrime.tf1.getText();           // 获取文本框中的内容
        String text2 = TestPrime.tf2.getText();
        String text3 = TestPrime.tf3.getText();
        int a, b, c;
        a = Integer.parseInt(text1);                       // 将字符串类型转换为整型
        b = Integer.parseInt(text2);
        c = Integer.parseInt(text3);
        boolean flag;
        int m, p, count = 0;
        for (m = a; m <= b; m++) {

```

```

        flag = true;
        for (p = 2; p <= m / 2; p++)                // 判断是否是质数
            if (m % p == 0) {
                flag = false;
                break;
            }
        if (flag) {
            String str = String.valueOf(m);          // 将整型转换为字符串类型
            TestPrime.ta.append(str + " ");          // 将质数写入到文本区中
            count++;
            TestPrime.num++;
            if (count % c == 0) {                    // 每行中只输出 c 个质数
                TestPrime.ta.append("\n");
            }
        }
    }
    String str = String.valueOf(TestPrime.num);
    TestPrime.tf4.setText(str);
}
}

```

运行程序，在文本框中依次输入“10、600、8”，单击“求 a 到 b 之间的质数”按钮，将在文本区中显示 10~600 之间的所有质数，并计算出质数个数，显示在下面的文本框中。程序运行结果如图 T9.2 所示。

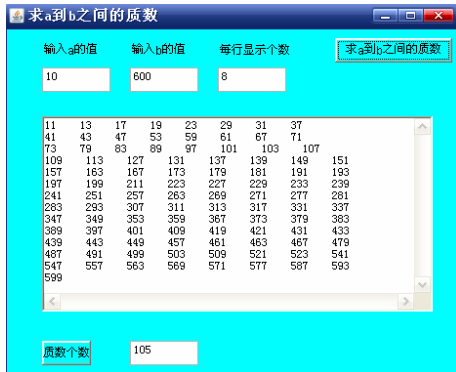


图 T9.2 计算 10~600 之间的质数

【实验 9.3】 使用两个文本区，在第一个文本区中输入一串无序的数值，对它们进行从小到大排序，将这些排好序的数值放到第二个文本区中。

ArraySort.java

```

package org.awt;
import java.awt.*;
import java.util.Arrays;
import java.util.regex.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

```

```

public class ArraySort extends Frame implements ActionListener{
    static TextArea ta1 = new TextArea();
    static TextArea ta2 = new TextArea();
    static Button bt1 = new Button("排序前");
    static Button bt2 = new Button("排序后");
    public static void main(String[] args) {
        ArraySort as = new ArraySort();
        as.setLayout(null);                // 取消布局管理器
        as.setBackground(Color.cyan);
        as.setSize(300, 200);
        as.setVisible(true);
        ta1.setBounds(70, 40, 220, 70);
        ta2.setBounds(70,120,220,70);
        as.add(ta1);
        bt1.setBounds(10,60,50,30);
        bt2.setBounds(10,140,50,30);
        bt1.setBackground(Color.cyan);
        bt2.setBackground(Color.cyan);
        as.add(bt1); as.add(bt2);
        bt2.addActionListener(as);        // 注册事件监听器
        as.add(ta2);
    }
    public void actionPerformed(ActionEvent e) {
        String s = ta1.getText();
        int[] arr = new int[10];          // 创建 1 个整型数组
        Pattern p = Pattern.compile("\\d{1,4}"); // 编译正则表达式
        Matcher m = p.matcher(s);        // 对字符串进行匹配
        int i =0;
        while(m.find()) {                // 寻找与指定模式匹配的下一个子序列
            int j = 0;
            j = Integer.parseInt(m.group()); // 将字符串类型转换为整型
            arr[i]= j;
            i++;
        }
        Arrays.sort(arr);                // 对数组进行排序
        for(int c = 0;c<i;c++){
            String str1 = String.valueOf(arr[c]);
            ta2.append(str1+"    ");      // 将数组中的内容输出到文本区中
        }
    }
}

```

运行程序，在窗体的第一个文本区中输入 10 个无序的数值，单击“排序后”按钮，这 10 个数值经过从小到大排序后，输出到第二个文本区中，如图 T9.3 所示。

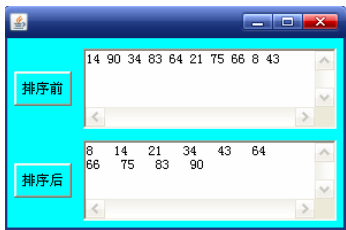


图 T9.3 对数值进行从小到大排序

思考与练习题

- (1) 用多行编辑框组件实现文本文件的编辑处理。在菜单上提供打开、保存、查找、替换等功能。
- (2) 编写一个程序，它包括一个文本框和三个按钮，单击每个按钮时，在文本框中显示不同的文字。
- (3) 使用两个文本区，在第一个文本区中输入一串无序的数值，对它们进行从大到小排序，将这些排好序的数值放到第二个文本区中。
- (4) 设计一个 Java 程序，在两个文本框中各放入一个日期值，计算这两个日期值之间间隔的天数，将计算的天数放到第三个文本框中。

实验 10 Swing 组件及应用

实验目的

- (1) 掌握常用的 Swing 组件。
- (2) 理解 AWT 事件处理原理。

实验准备

- (1) 复习本书第 10 章的有关内容。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 10.1】 在两个文本框里各输入一个整数，在第三个文本框里显示这两个数的最大公约数和最小公倍数。

JTextFieldDemo.java

```
package org.swing;
import java.awt.*;
```

```

import java.awt.event.*;
import javax.swing.*;

public class JtextFieldDemo extends JFrame{
    static JtextField tf1 = new JtextField();
    static JtextField tf2 = new JtextField();
    static JtextField tf3 = new JtextField();
    public JtextFieldDemo(){
        Container contentPane = getContentPane(); // 返回此窗体的 contentPane 对象
        contentPane.setLayout(null); // 取消布局管理器
        contentPane.setBackground(Color.cyan); // 设置窗口的颜色
        setLocation(300, 300);
        setSize(new Dimension(300, 200));
        JLabel l1 = new JLabel("第一个数");
        JLabel l2 = new JLabel("第二个数");
        JLabel l3 = new JLabel("所求的数");
        Button bt1 = new Button("求公约数和公倍数");
        bt1.setBackground(Color.cyan);
        bt1.addActionListener(new NumAction()); // 注册事件监听器
        bt1.setBounds(new Rectangle(80, 120, 120, 25));
        l1.setBounds(new Rectangle(30, 20, 60, 25));
        l2.setBounds(new Rectangle(120, 20, 60, 25));
        l3.setBounds(new Rectangle(210, 20, 120, 25));
        tf1.setBounds(new Rectangle(30, 50, 70, 25));
        tf2.setBounds(new Rectangle(120, 50, 70, 25));
        tf3.setBounds(new Rectangle(210, 50, 70, 25));
        tf3.setEditable(false);
        setVisible(true); // 设置窗体可见
        contentPane.add(l1);contentPane.add(l2);contentPane.add(l3);
        contentPane.add(bt1);
        contentPane.add(tf1);contentPane.add(tf2);contentPane.add(tf3);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 关闭窗口
    }
    public static void main(String[] args) {
        JtextFieldDemo fd = new JtextFieldDemo();
    }
}

class NumAction implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        int m ,n, r =0;
        int gcd = 0; // 最大公约数
        int lcm =0; // 最小公倍数
        m = Integer.parseInt(JtextFieldDemo.tf1.getText()); // 把字符串转换为整型
        n = Integer.parseInt(JtextFieldDemo.tf2.getText());
        lcm = m * n ;
        while((r=m%n)!=0){
            m = n;
            n =r;
        }
    }
}

```

```

    }
    gcd = n;
    lcm = lcm / gcd;
    String str1 = String.valueOf(gcd);           // 返回此字符串形式
    String str2 = String.valueOf(lcm);
    String s = str1+"    "+str2;
    JTextFieldDemo.tf3.setText(s);               // 设置第三个文本框的内容
}
}

```

运行程序，在第一个文本框里输入 9，在第二个文本框里输入 15，单击“求公约数和公倍数”按钮，则这两个数的最大公约数和最小公倍数显示在第三个文本框里。程序运行结果如图 T10.1 所示。

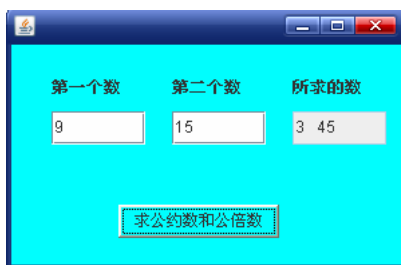


图 T10.1 文本框界面

【实验 10.2】 在窗体上创建三个文本框，两个文本框用于运算对象，另一个用于存放计算结果，下拉列表框存放四则运算符号。

MultiplyOperation.java

```

package org.swing;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MultiplyOperation extends JFrame{
    JTextField num1;
    JTextField num2;
    JTextField sum;
    static Choice ch = new Choice();
    public static void main(String[] args) {
        MultiplyOperation test = new MultiplyOperation();
        test.operation();
        test.getContentPane().setBackground(Color.cyan);           // 设置窗体的颜色
        test.setSize(280, 150);
        test.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);       // 关闭窗口
    }
}

```



```

public void operation() {
    num1 = new JTextField();
    num2 = new JTextField();
    sum = new JTextField();
    ch.add("+");
    ch.add("-");
    ch.add("*");
    ch.add("/");
    num1.setColumns(5); // 设置此文本框的列数
    num2.setColumns(5);
    sum.setColumns(5);
    setLayout(new FlowLayout()); // 采用流式布局管理器
    Button btnEqual = new Button("=");
    btnEqual.setBackground(Color.cyan);
    btnEqual.addActionListener(new MyListener(this)); // 注册事件监听器
    ch.addItemListener(new ChoiceHandler()); // 注册事件监听器
    add(num1); // 将文本框加入到窗体上
    add(ch);
    add(num2);
    add(btnEqual);
    add(sum);
    setVisible(true); // 设置窗体可见
}
}

class MyListener implements ActionListener {
    private MultiplyOperation mulp;
    public MyListener(MultiplyOperation mulp) {
        this.mulp = mulp;
    }
    public void actionPerformed(ActionEvent e) {
        String s1 = mulp.num1.getText(); // 获取文本框中的内容
        String s2 = mulp.num2.getText();
        int i1 = Integer.parseInt(s1); // 将字符串类型转换为整型
        int i2 = Integer.parseInt(s2);
        String itm;
        itm = ChoiceHandler.itm1 ;
        if (itm.equals("+")) {
            mulp.sum.setText(String.valueOf(i1 + i2));
        } else if (itm.equals("-")) {
            mulp.sum.setText(String.valueOf(i1 - i2));
        } else if (itm.equals("*")) {
            mulp.sum.setText(String.valueOf(i1 * i2));
        } else if (itm.equals("/")) {
            mulp.sum.setText(String.valueOf(i1 / i2));
        }
    }
}
}

```

```
class ChoiceHandler implements ItemListener {
    static String itm1;
    public void itemStateChanged(ItemEvent e) {
        itm1 = MultiplyOperation.ch.getSelectedItem();    // 获得所选项
    }
}
```

运行程序，在文本框中输入 5 和 6，单击等号按钮，两数相乘的积存放在第三个文本框中。程序的运行结果如图 T10.2 所示。

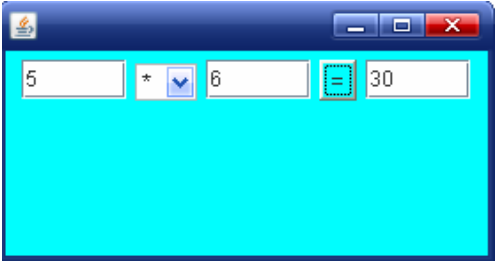


图 T10.2 计算两数相乘的积

思考与练习题

- (1) 从 `JButton` 继承编写一个新的按钮，每当按钮按下时，将为按钮随机选择一种颜色。
- (2) 编写一个应用程序，创建一个复选框捕获其事件，并在事件处理程序中向文本框中插入不同的文字。
- (3) 设计一个窗体，上面有一个按钮。当鼠标移到按钮上时，立即隐藏该按钮；当鼠标离开按钮时，显示该按钮。
- (4) 使用文本框和文本区，在文本框里输入要输出的杨辉三角形的行数，在文本区里显示杨辉三角形。

实验 11 并 发

实验目的

- (1) 掌握两种创建多线程的方法。
- (2) 了解线程的并发与互斥。

实验准备

- (1) 复习本书第 11 章的有关内容。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 11.1】 通过继承 Thread 类来创建多线程。

ThreadTest.java

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        new SimpleThread("first thread is running.").start();           // 启动线程
        new SimpleThread("second thread is running.").start();
    }
}
```

程序运行结果:

```
0 first thread is running.
0 second thread is running.
1 first thread is running.
2 first thread is running.
3 first thread is running.
4 first thread is running.
1 second thread is running.
5 first thread is running.
6 first thread is running.
```

【实验 11.2】 一个线程优先级的程序。在程序中，通过 `getPriority()` 方法获得现有线程的优先级，并且可以通过 `setPriority()` 方法来修改线程的优先级。

SimplePriorities.java

```
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
    private int countDown = 5;
    private volatile double d;
    private int priority;
    public SimplePriorities(int priority) {
        this.priority = priority;
    }
}
```

```

    }
    public String toString() {
        return Thread.currentThread() + ": " + countDown;
    }
    public void run() {
        Thread.currentThread().setPriority(priority);           // 设置线程的优先级
        while (true) {
            for (int i = 1; i < 100000; i++) {
                d += (Math.PI + Math.E) / (double) i;
                if (i % 1000 == 0)
                    Thread.yield();                             // 线程让步
            }
            System.out.println(this);
            if (--countDown == 0)
                return;
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i = 0; i < 5; i++)
            exec.execute(new SimplePriorities(Thread.MIN_PRIORITY)); // 最低优先级
        exec.execute(new SimplePriorities(Thread.MAX_PRIORITY));     // 最高优先级
        exec.shutdown();
    }
}

```

程序运行结果:

```

main start
main
JoinTest:0
JoinTest:1
JoinTest:2
JoinTest:3
JoinTest:4
JoinTest:5
JoinTest:6
JoinTest:7
JoinTest:8
JoinTest:9
main end

```

【实验 11.3】 创建一个实现睡眠（睡眠时间在 1~10s）功能的线程，显示它的睡眠时间并退出。

Test.java

```

public class Test {
    public static void main(String args[]) throws Exception {
        JoinTest jt = new JoinTest();
        jt.setName("JoinTest:");
        jt.start();                                           // 启动 JoinTest 线程
        System.out.println("main start");
        System.out.println(Thread.currentThread().getName()); // 获取正在运行的线程名
    }
}

```

```

        jt.join();
        System.out.println("main end");
    }
}
class JoinTest extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread().getName()+i);
        }
    }
}

```

程序运行结果:

```

main start
main
JoinTest:0
JoinTest:1
JoinTest:2
JoinTest:3
JoinTest:4
JoinTest:5
JoinTest:6
JoinTest:7
JoinTest:8
JoinTest:9
main end

```

思考与练习题

- (1) 创建一个线程，它将睡眠 1~10s 之间的随机数量的时间，然后显示它的睡眠时间并退出。
- (2) 创建三个线程，使得一个线程的优先级最高，一个线程的优先级最低，另一个线程的优先级位于两者之间。
- (3) 创建一个线程，使得当前线程调用 `yield()` 方法暂时放弃 CPU，给其他线程运行的机会。

实验 12 综合实例

实验目的

- (1) 综合应用第 1 章~第 11 章的知识。
- (2) 通过汉诺塔，锻炼解决问题的能力。

实验准备

- (1) 复习本书第 1 章~第 11 章的有关内容。

(2) 在阅读第 12 章的所有内容的基础上进行消化和理解。

实验内容

- (1) 按照书上的步骤编写代码，创建汉诺塔及实现手动搬运盘子。
- (2) 按照书上的步骤编写代码，创建汉诺塔及实现自动搬运盘子。

思考与练习题

看看有没有别的方法，实现相同的功能，并且修改原来的程序实现它。

实验 13 Java网络编程

实验目的

- (1) 掌握 TCP 和 UDP 网络编程。
- (2) 了解 URL 编程。

实验准备

- (1) 复习本书第 13 章的有关内容。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 13.1】 客户端程序向服务器程序发送任意的字符串，服务器程序收到后，显示收到的字符串，并统计字符串的长度。

TCP 服务器端 TCPServer.java

```
package org;
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class TCPServer {
    static DataInputStream dis = null;
    public static void main(String[] args) {
        boolean started = false;
        Socket s = null;
        TextArea ta = new TextArea();
```

```

ta.append("从客户端接收的数据: "+"\\n");
ServerSocket ss = null;
try {
    ss = new ServerSocket(8800);                // 创建一个监听 Socket 对象
} catch (BindException e) {
    System.exit(0);
} catch (IOException e) {
    e.printStackTrace();
}
Frame f = new Frame("服务器端");
f.setLocation(300, 300);
f.setSize(200, 200);
f.add(ta, BorderLayout.NORTH);
f.pack();
f.addWindowListener(new WindowAdapter() {      // 关闭窗口
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
f.setVisible(true);                            // 设置窗体可见
try {
    started = true;
    while (started) {
        boolean bConnected = false;
        s = ss.accept();                        // 等待客户端请求连接
        bConnected = true;
        dis = new DataInputStream(s.getInputStream());
        while (bConnected) {
            String str = dis.readUTF();          // 从输入流中读取数据
            str.length();
            ta.append(str+"", "字符串长度: "+str.length()+"\\n"); // 将数据添加到文本
                                                                    区中
        }
    }
} catch (EOFException e) {
    System.out.println("Client closed!");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (dis != null)
            dis.close();                        // 关闭输入流
        if (s != null)
            s.close();                          // 关闭 Socket 对象
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

```

    }
}
}

```

TCP 客户端 TCPClient.java

```

package org;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class TCPClient extends Frame {
    Socket s = null;
    DataOutputStream dos = null;
    DataInputStream dis = null;
    TextField tf = new TextField(40);
    List list = new List(6);
    public static void main(String[] args) {
        TCPClient client = new TCPClient();
        client.list.add("向服务器端发送的数据: ");
        client.setTitle("客户端");
        client.run();
    }
    public void run() {
        setLocation(400, 300);           // 设置窗体的位置
        this.setSize(300, 300);          // 设置窗体的大小
        add(tf, BorderLayout.SOUTH);
        add(list, BorderLayout.NORTH);
        pack();
        this.addWindowListener(new WindowAdapter() { // 关闭窗体
            public void windowClosing(WindowEvent e) {
                disconnect();
                System.exit(0);
            }
        });
        tf.addActionListener(new MyListener()); // 注册事件监听器
        setVisible(true);
        connect();
    }
    public void connect() {
        try {
            s = new Socket("127.0.0.1", 8800); // 创建一个向服务器发起连接的 Socket 对象
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```



```

public void disconnect() {
    try {
        dos.close();           // 关闭输出流
        s.close();             // 关闭 Socket 对象
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String s1 = null;
        String s2 = null;
        String str = tf.getText().trim();           // 获取文本框中的内容
        list.add(str);
        tf.setText("");                             // 将文本框的内容清空
        try {
            dos = new DataOutputStream(s.getOutputStream());
            dos.writeUTF(str);                       // 向流中写入数据
            dos.flush();                             // 刷新输出流
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
}
}

```

运行上面的 TCP 客户服务器程序。首先在客户端文本框中输入字符串，按下回车键，这些字符串将显示在客户端的窗口上。并且，这些数据将发送给服务器端，服务器端接收到从客户端发送的字符串，将显示在窗口上，如图 T13.1、图 T13.2 所示。



图 T13.1 服务器端

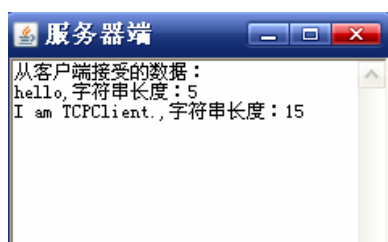


图 T13.2 客户端

思考与练习题

- (1) 设计一个通过 TCP 服务器转交方式，两个客户机之间一对一的网络通信程序。
- (2) 设计一个通过 UDP 服务器转交方式，两个客户机之间一对一的网络通信程序。
- (3) 编写一个通过返回与域名相关联的 IP 地址。

实验 14 JDBC编程

实验目的

- (1) 掌握 JDBC 编程。
- (2) 熟悉批处理和事务处理。

实验准备

- (1) 复习本书第 14 章的有关内容。
- (2) 进入 Eclipse 集成开发环境。

实验内容

【实验 14.1】 设计一个简单的 JDBC 程序。这里使用本书第 14 章所创建的数据库 xscj 和表 xs。

TestJDBC2.java

```
package org.jdbc;
import java.sql.*;
public class TestJDBC2 {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        try {
            /*******加载并注册 MySQL 的 JDBC 驱动*****/
            Class.forName("com.mysql.jdbc.Driver");
            /*******建立到 MySQL 数据库的连接*****/
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            /*******访问数据库，执行 SQL 语句*****/
            stmt = conn.createStatement();
            /*******添加记录*****/
            System.out.println("添加记录后:");
            String sql1 = "insert into xs values (100014, '邵辉', '土木工程')";
            stmt.executeUpdate(sql1);
            rs = stmt.executeQuery("select * from xs");
            while(rs.next()) {
                System.out.println(rs.getInt("id"));
                System.out.println(rs.getString("name"));
                System.out.println(rs.getString("profession"));
            }
        }
    }
}
```

```

/*****修改记录*****/
System.out.println("修改记录后:");
String sql2 = "update xs set name = '王涛' where id = 100014";
stmt.executeUpdate(sql2);
rs = stmt.executeQuery("select * from xs");
while(rs.next()) {
    System.out.println(rs.getInt("id"));
    System.out.println(rs.getString("name"));
    System.out.println(rs.getString("profession"));
}
System.out.println("删除记录后:");
/*****删除记录*****/
String sql3 = "delete from xs where id = 100014";
stmt.executeUpdate(sql3);
rs = stmt.executeQuery("select * from xs");
while(rs.next()) {
    System.out.println(rs.getInt("id"));
    System.out.println(rs.getString("name"));
    System.out.println(rs.getString("profession"));
}
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if(rs != null) {
            rs.close();           // 关闭 ResultSet 对象
            rs = null;
        }
        if(stmt != null) {
            stmt.close();         // 关闭 Statement 对象
            stmt = null;
        }
        if(conn != null) {
            conn.close();         // 关闭 Connection 对象
            conn = null;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}

```

程序运行结果:

添加记录后：
100014
邵辉
土木工程
修改记录后：
100014
王涛
土木工程
删除记录后：

【实验 14.2】 在 Java 程序中创建数据库和表，并使用批处理和事务处理。

UseJDBC2.java

```
package org.jdbc;
import java.sql.*;
public class UseJDBC2 {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        PreparedStatement pstmt = null;
        try {
            /*****加载并注册 MySQL 的 JDBC 驱动*****/
            Class.forName("com.mysql.jdbc.Driver");
            /*****建立到 MySQL 数据库的 URL*****/
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/mysql", "root", "123456");
            conn.setAutoCommit(false);
            /*****访问数据库，执行 SQL 语句*****/
            stmt = conn.createStatement();
            stmt.executeUpdate("create database bookstore"); // 创建数据库
            stmt.executeUpdate("use bookstore"); // 使 bookstore 成为当前数据库
            /*****创建表结构*****/
            stmt.executeUpdate("create table books(id int(4) not null primary key
                ,title char(20) not null,price float(4,2) not null)");
            /*****加入 sql 命令到命令列表中*****/
            stmt.addBatch("insert into books values (1001, 'Java 实用教程',39.00)");
            stmt.addBatch("insert into books values (1002, 'Jsp 网站编程',49.00)");
            stmt.addBatch("insert into books values (1003, 'Struts2 核心编程',58.00)");
            stmt.addBatch("insert into books values (1004, 'Hibernate 必备宝典 ',89.00)");
            stmt.addBatch("insert into books values (1005, 'C 程序设计',35.00)");
            stmt.executeBatch(); // 批量处理
            conn.commit(); // 提交事务
            rs = stmt.executeQuery("select * from books");
            while(rs.next()) {
                System.out.print(rs.getInt("id"));
                System.out.print(rs.getString("title"));
                System.out.print(rs.getString("price"));
                System.out.println();
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```

    } catch (SQLException e) {
        e.printStackTrace();
        try {
            if (conn != null) {
                conn.rollback();           // 事务回滚
                conn.setAutoCommit(true);  // 开启自动提交
            }
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    } finally {
        try {
            if(rs != null) {
                rs.close();                 // 关闭 ResultSet 对象
                rs = null;
            }
            if(stmt != null) {
                stmt.close();               // 关闭 Statement 对象
                stmt = null;
            }
            if(conn != null) {
                conn.close();               // 关闭 Connection 对象
                conn = null;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
}

```

程序运行结果:

```

1001Java实用教程39.00
1002Jsp网站编程49.00
1003Struts2核心编程58.00
1004Hibernate必备宝典69.00
1005C程序设计35.00

```

思考与练习题

- (1) 设计一个程序,实现对表的数据处理,如增加、修改、删除、查询等功能。
- (2) 设计一个程序,使用 JDBC 的事务处理功能。
- (3) 设计一个程序,使用 JDBC 的事务处理和批处理功能。

第 4 部分 习题答案

第 1 章 习题答案

1. Java 技术分为三大体系, 分别是 J2ME (Java 2 Micro Edition)、J2SE (Java 2 Standard Edition)、J2EE (Java 2 Enterprise Edition)。J2ME 用于嵌入式应用的 Java 2 平台, J2SE 用于工作站、PC 的 Java 2 标准平台, J2EE 用于可扩展的企业级应用的 Java 2 平台。

2. Java 语言的主要特点有: 简单性、面向对象、分布性、安全性、可移植性、高性能、多线程。

3. Java 解释器在没有生成任何实例的情况下, 以 `main()` 方法作为入口来执行程序。也可以在一个类中调用另一个类中的 `main()` 方法。

4. 用 “`javac`” 命令编译 Java 程序, 用 “`java`” 命令运行 Java 程序。

5. 程序源码:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.print("HelloWorld!");  
    }  
}
```

程序运行结果:

```
hello world!
```

第 2 章 习题答案

1. “工资” 是合法的标识符, 汉字属于 Java 中 “字母” 的范畴, 但不建议使用; “_123” 是合法的标识符, 标识符能以 “_” 开头; “4foots” 不是合法的标识符, 标识符不能以数字开头; “\$123” 是合法的标识符; “12a_2xy” 不是合法的标识符, 不能以数字开头; “exam-1” 不是合法的标识符, 标识符中不允许出现 “-” 号; “x+y” 不是合法的标识符, 不能出现 “+” 号。

2. Java 中基本数据类型包括: `boolean` (布尔型)、`byte` (字节型)、`short` (短整型)、`int` (整型)、`long` (长整型)、`char` (字符型)、`float` (单精度浮点型)、`double` (双精度浮点型) 共八种, 它们在内存中分别占用的字节数为未定义、1 字节、2 字节、4 字节、8 字节、2 字节、4 字节、8 字节。

3. (1) $a * x * x + b * x + c$ (2) $(x + y) * (x + y) * (x + y)$ (3) $(a + b)(a + b) / (a - b)$

4. (1) 3.5 (2) 14.0

5. (1) 10 (2) 0 (3) 0 (4) 10

6. 程序源码:

```
public class Enumerate {
    public static void main(String[] args) {
        int m = 0; int s = 0;
        int i = 0;
        for (m = 2; m < 1000; m++) {
            s = 0;
            for (i = 1; i < m; i++)
                if ((m % i) == 0)
                    s = s + i;           // 计算因子之和
            if (s == m) {                // 判断该数的因子之和是否等于该数自身
                System.out.print(m+"\t"+"它的完数是: ");
                for (i = 1; i < m; i++) {
                    if (m % i == 0) {
                        System.out.print(i + " "); // 输出完数的各个因子
                    }
                }
                System.out.println();
            }
        }
    }
}
```

程序运行结果:

6	它的完数是:	1	2	3															
28	它的完数是:	1	2	4	7	14													
496	它的完数是:	1	2	4	8	16	31	62	124	248									

7. 程序源码:

```
public class TestInvert {
    /*******返回 n 的反序数, 若 n=1234, 则返回 4321*****
    static int reverse(int n, int v) {
        return n == 0 ? v : reverse(n / 10, v * 10 + n % 10);
    }
    public static void main(String[] args) {
        System.out.println("1234 的反序打印是: " + reverse(1234, 0));
    }
}
```

程序运行结果:

1234的反序打印是: 4321

8. 程序源码:

```
public class Recurrence {
    public static void main(String[] args) {
        int k = Integer.parseInt(args[0]);           // 把字符串转换为整型
        long num = 0; int s = 1;
        for(int i=1;i<=k;i++){
            s *=i;                                     // s=i!
            num += s;                                   // num = 1!+ ... +i!
        }
    }
}
```

```
System.out.println("1!+2!+3!"+"..."+k+"!="+num);
```

```
// 打印乘积的累加和
```

```
}
```

```
}
```

右击“Recurrence.java”，选择“Run As”→“Run Configurations”，如图 S2.1 所示，选择“Main”标签页，在“Project”栏中选择“MyProject_02”，在“Main class”栏中选择“Recurrence”，选择“Arguments”标签页，在“Program arguments”栏中输入“5”，然后单击“Run”按钮，运行程序。

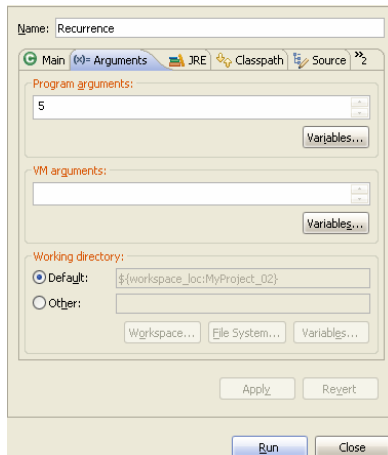


图 S2.1 计算阶乘的累加和

程序运行结果：

```
1!+2!+3!+...5!=153
```

第 3 章 习题答案

1. 在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义位于方法定义之间，仍旧会在方法（包括构造方法）被调用之前得到初始化。先初始化静态数据，然后才初始化非静态数据。

2. 无论创建多少个对象，静态数据在内存中只有一份拷贝，并且在不创建对象时，就可以使用静态数据。对于非静态数据，创建多少个对象，在内存中就有多少份拷贝。在未创建对象时不能直接使用非静态数据。

3. 当类中有两个同名变量，一个属于类的成员变量，另一个属于某个特定的方法（方法中的局部变量）时，可使用 **this** 区分成员变量和局部变量。使用 **this** 还可以简化构造方法的调用。

4. 程序源码：

```
public class TestInitialize {  
    public static void main(String[] args){  
        Tester t = new Tester();  
        System.out.println(t.s);  
    }  
}
```



```

}
class Tester {
    String s;
}

```

程序运行结果:

```

null

```

5. 程序源码:

```

public class BuildObject {
    BuildObject(){
        System.out.println("invoke BuildObject!");
    }
    public static void main(String[] args){
        BuildObject object = new BuildObject();
    }
}

```

程序运行结果:

```

invoke BuildObject!

```

第 4 章 习题答案

1. 如果一个抽象类中的所有方法都是抽象的, 那么这个抽象类实际上就是一个接口, 即一种特殊的抽象类。由 `abstract` 修饰的类称为抽象类, 只声明而没有实现的方法称为抽象方法。

2. 继承是复用程序代码的有力手段。当多个类之间存在相同的属性和方法时, 可以把这些相同的属性和方法抽取出来放到一个单独的类中, 这个单独的类被定义为父类 **Base** 类。其他的类通过 `extends` 关键字来声明继承父类 **Base** 类, 则这些类自动拥有父类 **Base** 类中的所有属性和方法, 继承父类的类被定义为子类 **Sub** 类。**Object** 类是所有类的祖先, 支持类的单一继承。一个类可实现多个接口, 接口可多重继承, 构造方法不能继承。

3. **Java** 运行时多态又叫做动态绑定, 就是在运行期间 (而非编译期间) 判断所引用对象的实际类型并根据对象的类型进行绑定, 从而调用恰当的方法。

4. 程序源码:

```

abstract class Dad {} // 接口
class Son1 extends Dad {
    protected void print() { System.out.println("Son"); }
}
abstract class SecondDad {
    abstract protected void print();
}
class SecondSon extends SecondDad {
    protected void print() { System.out.println("SecondSon"); }
}

```

```

public class Polymorphism {
    public static void testPrint(Son1 s) {
        ((Son1)s).print();
    }
    public static void secondTestPrint(SecondDad sd) {
        sd.print();
    }
    public static void main(String[] args) {
        Son1 s = new Son1();
        Polymorphism.testPrint(s);
        SecondSon ss = new SecondSon();
        Polymorphism.secondTestPrint(ss);
    }
}

```

// 静态方法

// 调用静态方法

程序运行结果：

```

Son
SecondSon

```

5. 程序源码：

```

package org.interface1;
public interface TestMethod {
    void studying(); void playing(); void eating();
}
package org.interface2;
import org.interface1.*;
public class Test implements TestMethod{
    public void studying(){
        System.out.println("I am studying hard.");
    }
    public void playing(){
        System.out.println("I like playing football.");
    }
    public void eating(){
        System.out.println("I like eating fish.");
    }
    public static void main(String [] args){
        Test test = new Test();
        test.studying();
        test.playing();
        test.eating();
    }
}

```

// 导入 org.interface1 包中的类

程序运行结果：

```

I am studying hard.
I like playing football.
I like eating fish.

```

第 5 章 习题答案

1. 所谓异常，就是以对象的方式表示的一个或一类错误，该异常对象不仅封装了错误信息，还包含了错误发生时的“上下文”信息。

2. 首先执行 try 语句块中的代码，若一切正常，则跳过 catch 语句块，执行 finally 语句块中的代码（若该部分没有缺省），执行完后，整个 try-catch-finally 语句才算执行完成。若执行 try 语句块时产生异常，则立即跳转到 catch 语句块。Java 虚拟机会把实际抛出的异常对象依次和各个 catch 语句块声明的异常类型匹配，如果异常对象为某个异常类型或其子类的对象，就执行这个 catch 语句块，而不会再执行其他的 catch 语句块，之后，跳转到 finally 语句块，再执行完 finally 语句块，该语句才算执行完成。

3. throws 子句与 throw 语句都用于 Java 的异常处理。throws 子句用于声明抛出异常，只是表明一个方法可能会抛出异常，而真正抛出异常的动作是由抛出异常语句 throw 语句来完成的。

4. Throwable 类或其子类的对象可以用 throw 语句抛出。

5. 如果程序出现的错误不会导致程序异常终止，可以用 if 语句来处理程序错误；如果程序出现的异常会导致程序意外终止，则使用 try-catch-finally 语句作为异常处理。

6. 程序源码：

```
import java.util.Arrays;
public class StringBufferDemo {
    public static void main(String[] args) {
        String s = "JavaavaJ";
        char[] str = s.toCharArray();           // 返回一个 char 数组
        boolean b = true;
        int i = str.length - 1;
        int k = i;
        for(int j = 0; j < k && b; j++, i--){
            if(str[j] != str[i])                 // 不相等，则不是回文，结束循环
                b = false;
        }
        if (b == true){System.out.println("字符串"+s+"是回文");}
        else{System.out.println("字符串"+s+"不是回文");}
    }
}
```

程序运行结果：

```
字符串JavaavaJ是回文
```

第 6 章 习题答案

1. 程序源码：

```
package org.arrays;
```

```

public class InvertSequence {
    public static void main(String[] args){
        int i=0;   int j=0;   int k=0;
        int l=0;   int m=0;   int n=0;
        int num=0;
        for(int p = 1001;p<10000;p++){
            i = p /1000;      j = p %1000;      k = j /100;
            l = j %100;      m = j /10;      n = j %10;
            num = n *1000+m*100+k*10+i*1;
            if (num ==p*9){
                System.out.println(p);
            }
        }
    }
}

```

程序运行结果：

```
1089
```

2. 程序源码：

```

package org.arrays;
import java.util.Arrays;
import java.util.Random;
public class ArraySortTest {
    public static void main(String[] args){
        Random r = new Random();
        int[] a = new int[10];
        System.out.println("排序前: ");
        for (int i=0;i<10;i++){
            a[i] = r.nextInt(100);
            System.out.print(a[i]+" ");
        }
        System.out.println();
        Arrays.sort(a);                                // 数组排序
        System.out.println("排序后: ");
        for (int i=9;i>=0;i--){
            System.out.print(a[i]+" ");
        }
    }
}

```

程序运行结果：

```

排序前:
59  46  78  43  46  19  79  91  57  92
排序后:
92  91  79  78  59  57  46  46  43  19

```

3. 程序源码：

```

package org.arrays;
public class MatrixAdd {

```

```

public static void main(String[] args) {
    int i, j;
    int[][] arr1 = new int[][]{{6,8,14},{7,5,6},{9,8,13}};
    int [][]arr2 = new int[][]{{3,4,6},{5,3,12},{12,11,21}};
    int[][] arr3= new int[3][3];
    for (i=0;i<3;i++){
        for(j=0;j<3;j++){
            arr3[i][j]=arr1[i][j]+arr2[i][j];
        }
    }
    for(int k =0;k<3;k++){
        for(int p=0;p<3;p++){
            System.out.print(arr3[k][p]+"");
        }
        System.out.println();
    }
}

```

程序运行结果:

9	12	20
12	8	18
21	19	34

4. 程序源码:

```

package org.enums;
public class Enum {
    public enum Bills {ONE, TWO,FIVE}
    public static void main(String[] args) {
        for(Bills b : Bills.values())
            System.out.println(b + ", ordinal " + b.ordinal());
    }
}

```

程序运行结果:

```

ONE, ordinal 0
TWO, ordinal 1
FIVE, ordinal 2

```

5. 程序源码:

```

package org.enums;
public class Wallet {
    Bills b;
    public static void main(String[] args) {
        for(Bills b : Bills.values()) {
            System.out.print("Worth: ");
            switch(b) {
                case ONE: System.out.println("¥1"); break;
                case FIVE: System.out.println("¥5"); break;
                case TEN: System.out.println("¥10"); break;
            }
        }
    }
}

```

```

        default: break;
    }
}
}
enum Bills {
    ONE, FIVE, TEN
}

```

程序运行结果:

```

Worth: ¥ 1
Worth: ¥ 5
Worth: ¥ 10

```

第 7 章 习题答案

1. Collection 是接口, Collections 是 java.util 包中的实用类, Collections 中提供了大量的静态方法实现对容器的搜索和排序等。

2. 在 Set 容器中, 用 equals()方法来区分元素是否重复。equals()方法用于判断一个对象是否等于另外一个对象, 实际上是比较两个引用是否指向同一个对象。操作符“==”用来比较两个操作元是否相等, 这两个操作元既可以是基本类型, 也可以是引用类型。当两个操作元都是引用类型时, 那么这两个引用变量必须都引用同一个对象才返回 true。

3. List、Set 接口继承自 Collection 接口, 而 Map 不继承自 Collection 接口。

4. 不对, 有相同的 hashCode。

5. 泛型引入的最主要原因就是安全地使用容器类。泛型实现了参数化类型的概念, 使代码可以应用于多种类型。

6. 程序源码:

```

package org.container;
import java.util.*;
public class UseCollections {
    public static void printAny(Collection c) {
        Iterator it = c.iterator();           // 返回一个迭代器
        while(it.hasNext())
            System.out.print(it.next() + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>(Arrays.asList(1, 2, 3,4,5));
        LinkedList<Character> ll =new LinkedList<Character>(Arrays.asList('a', 'b', 'c','d','e'));
        HashSet<Float> hs = new HashSet<Float>(Arrays.asList(10.1f, 20.2f, 30.3f, 40.3f,50.3f));
        TreeSet<Double> ts =new TreeSet<Double>(Arrays.asList(1.34, 2.34, 3.34,4.34,5.34));
        LinkedHashSet<Integer> lhs =new LinkedHashSet<Integer>(Arrays.asList(10, 20, 30,40,50));
        printAny(al);    printAny(ll);    printAny(hs);
        printAny(ts);    printAny(lhs);
    }
}

```

```
}  
}
```

程序运行结果:

```
1 2 3 4 5  
a b c d e  
30.3 10.1 40.3 50.3 20.2  
1.34 2.34 3.34 4.34 5.34  
10 20 30 40 50
```

7. 程序源码:

```
package org.container;  
import java.util.*;  
class Season {  
    private int num;  
    public Season(int i)  
    {num = i;}  
}  
public class UseHashMap {  
    public static void main(String[] args) {  
        Map<String, Season> map = new HashMap<String, Season>();  
        map.put("Spring", new Season(1));           // 加入元素到 HashMap 中  
        map.put("Summer", new Season(2));  
        map.put("Autumn", new Season(3));  
        map.put("Winter", new Season(4));  
        Set<String> sortedKeys = new TreeSet<String>(map.keySet()); // 排序  
        System.out.println(sortedKeys);  
        System.out.println();  
        Map<String, Season> sorted = new HashMap<String, Season>();  
        for(String s : sortedKeys) {  
            System.out.print("Adding " + s + ", ");  
            sorted.put(s, map.get(s));  
        }  
    }  
}
```

程序运行结果:

```
[Autumn, Spring, Summer, Winter]  
Adding Autumn, Adding Spring, Adding Summer, Adding Winter,
```

第 8 章 习题答案

1. 将对象保存起来以便程序下一次运行时再从外存将这些对象读入到内存中, 重建这些对象或者将对象通过网络传递到另一端的 Java 程序, 实施对象的这种操作称为对象的序列化(或持久化)。要进行对象序列化, 首先要创建某些 `OutputStream` 对象, 然后将其包装在一个 `ObjectOutputStream` 对象内。这时, 只需调用 `writeObject()` 即可将对象序列化, 并将其发

送给 `OutputStream`。要反向该过程（将一个序列还原为一个对象），需要将一个 `InputStream` 包装在 `ObjectInputStream` 内，再调用 `readObject()`。

2. 在 Java 程序中，定制对象的序列化，必须明确实现 `java.io.Serializable` 接口，声明自己可以序列化。

3. 程序源码：

```
package org.iostream;
import java.io.*;
public class FileCharacterStreamDemo {
    public static void main(String[] s) throws Exception {
        String filename1 = "e:/workbench/MyProject_08/src/org/iostream/t7.txt";
        String filename2 = "e:/workbench/MyProject_08/src/org/iostream/t8.txt";
        FileReader filereader = new FileReader(filename1);
        FileWriter filewriter = new FileWriter(filename2);
        int c;
        while ((c = filereader.read()) != -1)           // 从 FileReader 类中读取一个字符
            filewriter.write(c);                       // 向 FileWriter 类中写入一个字符
        filereader.close();
        filewriter.close();
    }
}
```

运行程序，t17.txt 文件中的内容将被复制到 t18.txt 文件中。

第 9 章 习题答案

1. `FlowLayout` 是把组件从左向右、从上向下，一个接一个地放到容器中，组件之间的默认间隔（水平和垂直）为 5 个像素，默认的组件对齐方式为居中。`BorderLayout` 将容器分成 5 个区域来安排组件：`North`、`South`、`East`、`West`、`Center`。`GridLayout` 将容器分成一个个格子，按行依次排列组件，各组件大小相同。`CardLayout` 将界面看做一系列卡片，在任何时候只有其中一张卡片是可见的，这张卡片占据容器的整个区域。如果不希望通过布局管理器来管理布局，可以调用容器的 `setLayout (null)` 方法，这样布局管理器就被取消了。

2. 容器是用来组织其他容器和基本组件的。一个容器可以容纳多个容器和基本组件，容器本身也是一个组件，具有组件的所有性质。

3. AWT 有 4 种实现监听接口的方式：内部类实现监听接口；类自身实现监听接口；外部类实现监听接口；采用事件适配器。

4. (1) (b)，(d) (2) (c) (3) (a) (4) (a)

5. (a)，(c)

6. 程序源码：

```
package org.awt;
import java.awt.*;
import java.awt.event.*;
public class TextAreaDemo {
```



```

private Frame f = new Frame("将单行编辑框中的内容添加到多行编辑框中");
private TextField tf = new TextField(20);
/*创建一个 TextArea 的对象 ta，有水平滚动条和垂直滚动条*/
private TextArea ta = new TextArea("", 5, 10, TextArea.SCROLLBARS_BOTH);
Font ft = new Font("Serif", Font.BOLD, 28);
public static void main(String args[]) {
    TextAreaDemo tad = new TextAreaDemo();
    tad.go();
}
void go() {
    f.setLayout(new BorderLayout(0, 10));
    f.add("North", tf);    f.add("Center", ta);
    ta.setFont(ft);        tf.setFont(ft);
    tf.addActionListener(new TextHandler());           // 注册监听器
    // 将窗口 f 注册到监听器 WindowHandler 上
    f.addWindowListener(new WindowHandler());          // 注册适配器
    f.setSize(450, 250);
    f.setResizable(true);
    f.setVisible(true);
}
// 监听器 TextHandler 实现了接口 ActionListener，必须实现方法 actionPerformed
class TextHandler implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        ta.append(tf.getText() + "\n");    // 文本框中的文字添加到文本区 ta 的后面
        tf.setText("");
    }
}
class WindowHandler extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(1);    // 关闭窗口
    }
}
}

```

运行程序，程序运行结果如图 S9.1 所示。

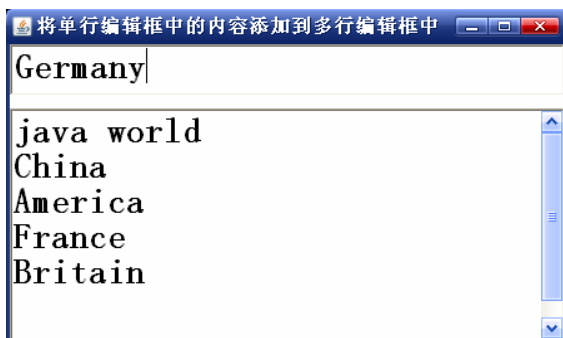


图 S9.1 使用 List

7. 程序源码:

```
package org.awt;
import java.awt.*;
import java.awt.event.*;
public class MyGridLayout extends Frame{
    private final String names[]={ "A","B","C","D","E","F","G","H","I","J","K","L"};
    private Button[] bts=new Button[12];
    private boolean flag=true;
    final GridLayout grid1=new GridLayout(3,4,5,10);
    final GridLayout grid2=new GridLayout(4,3);
    public MyGridLayout(String title){
        super(title);
        setBackground(Color.orange);
        setLayout();
        ActionListener listener=new ActionListener(){
            public void actionPerformed(ActionEvent event){
                setLayout();                // 切换网格布局
                MyGridLayout.this.validate();    // 使新的布局生效
            }
        };
        for(int i=0;i<bts.length;i++){
            bts[i]=new Button(names[i]);
            bts[i].addActionListener(listener);
            add(bts[i]);
        }
        setSize(300,300);
        setVisible(true);
    }
    //*****切换窗体布局管理器*****
    private void setLayout(){
        if(flag)
            setLayout(grid1);
        else
            setLayout(grid2);
        flag=!flag;
    }
    public static void main(String args[]){
        new MyGridLayout("This is a GridLayout");
    }
}
```

运行结果如图 S9.2 所示。

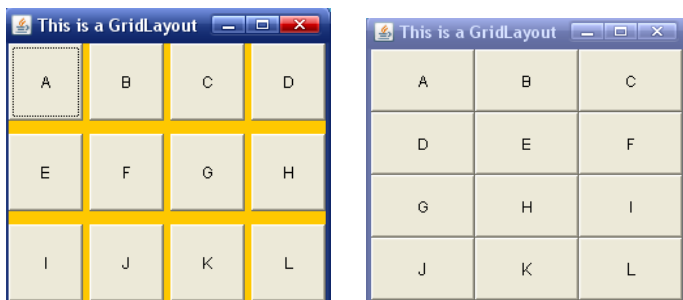


图 S9.2 采用 GridLayout 布局组件

第 10 章 习题答案

1. (1) (c) (2) (a) (3) (b) (4) (b)

2. 基本 AWT 库将处理用户界面元素的任务委派给每个目标平台（Windows、Solaris、Macintosh 等），由本地 GUI 工具箱负责用户界面元素的创建和动作。Swing 组件几乎都是轻量组件，它采用了与 AWT 完全不同的工作方式，它将按钮、菜单这样的用户界面元素绘制在空白窗口上，而对等体只需要创建和绘制窗口。

3. 程序源码：

```
package org.swing;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ListSwitchDemo {
    JFrame f = new JFrame("交换两个列表框的项目");
    JPanel p1 = new JPanel();
    JPanel p2 = new JPanel(new GridLayout(2, 1, 5, 30));
    JPanel p3 = new JPanel();
    Button b1 = new Button(">>");
    Button b2 = new Button("<<");
    List l1 = new List(10,true);
    List l2 = new List(10,true);
    Font ft = new Font("Serif", Font.BOLD, 18);
    public static void main(String args[]) {
        ListSwitchDemo ls = new ListSwitchDemo();
        ls.go();
    }
    void go() {
        f.setSize(500, 300);
        f.setLayout(new FlowLayout()); // 设置窗体的流式布局管理器
        f.add(p1);f.add(p2);f.add(p3); // 把 JPanel 对象加入到容器中
        l1.add("desk");l1.add("computer");l1.add("printer");//向列表框中加入项目
        l1.add("pen");l1.add("book");l1.add("paper");
        l1.setFont(ft); // 设置列表框中项目的字体
    }
}
```

```

l2.add("yellow");l2.add("brown");l2.add("white");
l2.add("pink");l2.add("red");l2.add("blue");
l2.setFont(ft);
p1.add(l1);p2.add(b1);p2.add(b2);
b1.setFont(ft);b2.setFont(ft);
p3.add(l2);
b1.addActionListener(new ButtonHandler(l1, l2)); // 注册事件监听器
b2.addActionListener(new ButtonHandler(l2, l1));
l1.addMouseListener(new MouseHandler(l1, l2));
l2.addMouseListener(new MouseHandler(l2, l1));
f.addWindowListener(new WindowHandler());
f.setVisible(true);
}

void exchange(List la, List lb) {
    String itm[];
    int idx[];
    itm = la.getSelectedItems();           // 获取选中的项
    idx = la.getSelectedIndexes();         // 获取选中的索引
    int i = idx.length;
    for(int k = 0; k < i; k++){
        lb.add(itm[k]);
        la.remove(idx[k]);
    }
}

class ButtonHandler implements ActionListener {
    List la, lb;
    ButtonHandler(List la, List lb) {
        this.la = la; this.lb = lb;
    }
    public void actionPerformed(ActionEvent e) { // 交换两个列表项的项目
        exchange(la, lb);
    }
}

class MouseHandler extends MouseAdapter {
    List la, lb;
    MouseHandler(List la, List lb) {
        this.la = la; this.lb = lb;
    }
    public void mouseClicked(MouseEvent e) {
        if (e.getClickCount() == 2)
            exchange(la, lb);
    }
}

class WindowHandler extends WindowAdapter { // 关闭窗口
    public void windowClosing(WindowEvent e) {
        System.exit(-1);
    }
}

```

```
}  
}
```

运行程序，单击左边列表框的选项，再单击“>>”按钮，则被选中的列表项就会移到右边的列表框中，运行结果如图 S10.1 所示。



图 S10.1 交换两个列表框的项目

4. 程序源码：

```
package org.swing;  
import java.awt.*;  
import javax.swing.*;  
public class MyJPanel {  
    public static void main(String args[]) {  
        JFrame fr = new JFrame();  
        fr.setTitle("JPanel 的用法");  
        BorderLayout bl = new BorderLayout();  
        fr.getContentPane().setLayout(bl);  
        JPanel jp = new JPanel(new GridLayout(2, 2));  
        fr.getContentPane().add("Center", jp);  
        JLabel jl = new JLabel("this is a test");  
        fr.getContentPane().add("South", jl);  
        JLabel l1 = new JLabel("First", JLabel.CENTER);  
        jp.add(l1);  
        JLabel l2 = new JLabel("Second", JLabel.CENTER);  
        jp.add(l2);  
        JLabel l3 = new JLabel("Third", JLabel.CENTER);  
        jp.add(l3);  
        Font ft = new Font("Serif", Font.BOLD, 18);  
        jl.setFont(ft); l1.setFont(ft);  
        l2.setFont(ft); l3.setFont(ft);  
        //设定窗口背景色为绿色  
        fr.getContentPane().setBackground(Color.green);  
        fr.setLocation(300, 500);  
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        fr.setSize(400, 300);  
        fr.setResizable(true);  
        fr.setVisible(true);  
    }  
}
```

```
}
```

运行程序，程序运行结果如图 S10.2 所示。



图 S10.2 使用 JPanel

第 11 章 习题答案

1. 程序是一组指令的有序集合，是一个静态的实体。进程是指运行中的应用程序，每一个进程都有自己独立的内存空间。一个进程可以由多个线程组成，即在一个进程中可以同时运行多个不同的线程，它们分别执行不同的任务。这些线程在同一块地址空间中工作，共享进程的状态和资源。

2. 创建线程有两种方式：第一种方式是通过继承 `java.lang.Thread` 类创建线程；第二种方式是通过实现 `Runnable` 接口创建线程。

3. 线程有四种状态：新建、就绪、阻塞和死亡状态。用 `new` 操作符创建一个线程对象时，此时线程处在新建（`new`）状态。其他线程调用它的 `start()` 方法，该线程就进入就绪状态。当线程因为某些原因放弃 CPU，暂时停止运行，此时线程处于阻塞状态。当线程退出 `run()` 方法时，就进入死亡状态，该线程结束生命周期。

第 13 章 习题答案

1. TCP 协议能为应用程序提供可靠的通信连接，使一台计算机发出的字节流无差错地发往网络上的其他计算机，对可靠性要求高的数据通信系统往往使用 TCP 协议传输数据。UDP 适用于一次只传送少量数据，对可靠性要求不高的应用环境。

2. 因为创建、启动和撤销一个线程都要消费资源，在 TCP 服务器多线程方式下，TCP 服务器开始启动时，立即创建 n 个线程。在整个 TCP 服务器工作期间，这 n 个线程一直在工作。当一个客户端发起连接时，若还有一个线程处理监听状态，则立即处理该客户端请求，并负责对该客户端本次的通信。当通信完成时，客户端离开，但该服务器线程并不死亡，而是继续等待下一个客户端的连接请求。

3. UDP 服务器采用循环方式好一些。因为 UDP 是面向无连接的，没有一个客户端可以独占服务器。只要处理过程不是死循环，服务器对于每一个客户端的请求都是能够处理的。但如果需要处理大量数据或长时间进行数据处理，则可以考虑使用多线程的方式。

4. `java.net.MulticastSocket` 类的对象，代表一个组播 `Socket`，可以发送和接收组播包。其实 `java.net.MulticastSocket` 类是 `java.net.DatagramSocket` 类的子类，只是增加了支持组播功能，因此 `DatagramSocket` 类中的方法都可以使用。组播包就是一个 **UDP** 数据报包。

5. 程序源码：

```
package com.net;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class InetAddressTest {
    static TextField tf1 = new TextField(40);
    static List list = new List(6);
    public static void main(String[] args) throws Exception {
        Frame f = new Frame();
        f.add(list);
        f.setSize(300, 300); // 设置窗体的大小
        Panel p = new Panel();
        p.setLayout(new BorderLayout()); // 设置边界布局管理器
        tf1.addActionListener(new MyListener()); // 注册事件监听器
        p.add("West", tf1);
        f.add("South", p);
        f.addWindowListener(new WindowAdapter() { // 关闭窗口
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        f.setVisible(true); // 设置窗体可见
    }
}
class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String s = InetAddressTest.tf1.getText(); // 获取文本框中的内容
        InetAddress[] addr;
        InetAddress addr2;
        try {
            InetAddressTest.list.removeAll(); // 将列表框中的原有内容清除
            addr = InetAddress.getAllByName(s); // 返回主机名所对应的所有 IP 地址
            addr2 = InetAddress.getLocalHost();
            for (int i = 0; i < addr.length; i++) {
                InetAddressTest.list.add(addr[i].toString()); // 添加到列表框中
            }
            InetAddressTest.list.add(addr2.toString()); // 将本地主机的 IP 地址添加到列表框里
        } catch (UnknownHostException e1) {
            e1.printStackTrace();
        }
        ((TextField) e.getSource()).setText(null); // 设置文本框的内容为空
    }
}
```

}

运行程序，在文本框中输入网易的域名“www.163.com”，则在列表框中显示网易的 IP 地址信息，并显示本地主机的 IP 地址，如图 S13.1 所示。

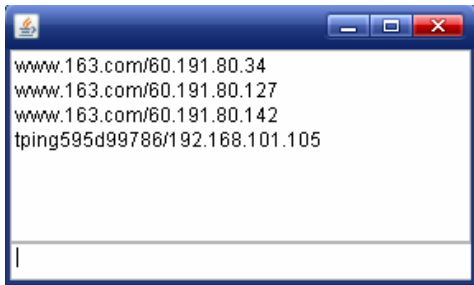


图 S13.1 获取网易的 IP 地址

第 14 章 习题答案

1. JDBC 驱动器有四种类型：JDBC-ODBC 驱动程序；本地代码和 Java 驱动程序；JDBC 网络纯 Java 驱动程序；本地协议的纯 Java 驱动程序。

2. (c)，(d)

3. (a)，(b)

4. (a)，(c)

5. (b)，(c)

6. 程序源码：

```
package org.jdbc;
import java.sql.*;
public class TestDMLDemo {
    public static void main(String[] args) {
        ResultSet rs = null;
        Statement stmt = null;
        Connection conn = null;
        try {
            Class.forName("com.mysql.jdbc.Driver"); // 加载并注册 MySQL 的 JDBC 驱动
            /*建立到 MySQL 数据库的连接*/
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/xscj", "root", "123456");
            stmt = conn.createStatement(); // 访问数据库，执行 SQL 语句
            System.out.println("添加记录后:");
            String sql1 = "insert into student values (100001, '李计', 1,80,78)";
            String sql2 = "insert into student values (100002, '王燕', 0,69,85)";
            stmt.executeUpdate(sql1);
            stmt.executeUpdate(sql2);
            rs = stmt.executeQuery("select * from student");
            while(rs.next()) {
```



```

        System.out.print("学号:"+rs.getInt("id")+" ");
        System.out.print("姓名:"+rs.getString("name")+" ");
        System.out.print("性别:"+rs.getString("sex"));
        System.out.print("数学成绩:"+rs.getString("math")+" ");
        System.out.print("英语成绩:"+rs.getString("english")+" ");
        System.out.println();
    }
    System.out.println("修改记录后:");
    String sql3 = "update student set name = '王涛' where id = 100002";
    stmt.executeUpdate(sql3);
    rs = stmt.executeQuery("select * from student");
    while(rs.next()) {
        System.out.print("学号:"+rs.getInt("id")+" ");
        System.out.print("姓名:"+rs.getString("name")+" ");
        System.out.print("性别:"+rs.getString("sex"));
        System.out.print("数学成绩:"+rs.getString("math")+" ");
        System.out.print("英语成绩:"+rs.getString("english")+" ");
        System.out.println();
    }
    System.out.println("删除记录后:");
    String sql4 = "delete from student where id = 100001";
    stmt.executeUpdate(sql4);
    rs = stmt.executeQuery("select * from student");
    while(rs.next()) {
        System.out.print("学号:"+rs.getInt("id")+" ");
        System.out.print("姓名:"+rs.getString("name")+" ");
        System.out.print("性别:"+rs.getString("sex"));
        System.out.print("数学成绩:"+rs.getString("math")+" ");
        System.out.print("英语成绩:"+rs.getString("english")+" ");
        System.out.println();
    }
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if(rs != null) {
            rs.close();                // 关闭 ResultSet 对象
            rs = null;
        }
        if(stmt != null) {
            stmt.close();              // 关闭 Statement 对象
            stmt = null;
        }
        if(conn != null) {
            conn.close();              // 关闭 Connection 对象

```

```
        conn = null;
    }
} catch (SQLException e) {
    e.printStackTrace();
}
}
}
```

程序运行结果:

添加记录后:

学号:100001 姓名:李计 性别:1数学成绩:80 英语成绩:78

学号:100002 姓名:王燕 性别:0数学成绩:69 英语成绩:85

修改记录后:

学号:100001 姓名:李计 性别:1数学成绩:80 英语成绩:78

学号:100002 姓名:王涛 性别:0数学成绩:69 英语成绩:85

删除记录后:

学号:100002 姓名:王涛 性别:0数学成绩:69 英语成绩:85

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为，歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036